Forecasting Avocado Prices Using ARIMA and FB Prophet

Aim/Goal: Forecast the prices of Avocado in the US

Data Source: Retailers' cash registers

Measure of Success: Since it's a regression task, we want to choose a model which has higher R^2 *score* and/or least *MAPE*. We will be using following data related metrics:

- R^2: Tells how well predictions approximate the real data points.
- RMSE: Standard deviation of prediction errors. Tells how concentrated the data is around line of best fit.
- MAPE: Average of the absolute percentage errors of forecasts.

Result: The prices needn't be forecasted on a live basis, so there is no need to deploy this model. Based on forecasted prices, stakeholders can decide about expanding their business to different types of Avocado farms. They can also focus on sales and price in each state and plan their strategy accordingly.

Kaggle Link: https://www.kaggle.com/datasets/neuromusic/avocado-prices

Data background and information (from Kaggle):

The data comes directly from retailers' cash registers based on the actual retail sales of Hass avocados.

- Data represents weekly retail scan data for National retail volume (units) and price from Apr 2015 to Mar 2018.
- The Average Price (of avocados) in the table reflects a per unit (per avocado) cost, even when multiple units (avocados) are sold in bags.
- The Product Lookup codes (PLU's) in the table are only for Hass avocados. Other varieties of avocados (e.g. greenskins) are not included in this table.

Some relevant columns in the dataset:

- Date - The date of the observation
- AveragePrice - The average price of a single avocado
- Type - conventional or organic
- Region - The city or region of the observation
- Total Volume - Total number of avocados sold
- 4046 - Total number of avocados with PLU 4046 sold
- 4225 - Total number of avocados with PLU 4225 sold
- 4770 - Total number of avocados with PLU 4770 sold
- Total Bags – Total bags sold
- Small/Large/XLarge Bags – Total bags sold by size

As mentioned above there are two types of avocados in the dataset as well as several different regions represented. All sorts of analysis for different areas of the United States, specific cities, or just the overall United States on either type of avocado is possible. The analysis will be focused on the complete dataset.

Code:

In [1]:

```
# Importing the required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.offline as py
import plotly.express as px
py.init_notebook_mode()
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")
```

In [4]:
```
plt.rcParams['figure.figsize'] = [15, 7]
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['text.color'] = 'GREEN'
```

In [5]:
```
# Read data
dataset = pd.read_csv('avocado.csv')
```

Understanding the data

Let's first look at the columns and the data:

In [6]:
```
dataset.head()
```

Out[6]:

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26

In [7]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18249 entries, 0 to 18248
Data columns (total 14 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Unnamed: 0    18249 non-null  int64  
 1   Date          18249 non-null  object 
 2   AveragePrice  18249 non-null  float64
 3   Total Volume  18249 non-null  float64
 4   4046          18249 non-null  float64
 5   4225          18249 non-null  float64
 6   4770          18249 non-null  float64
 7   Total Bags    18249 non-null  float64
 8   Small Bags    18249 non-null  float64
 9   Large Bags    18249 non-null  float64
 10  XLarge Bags   18249 non-null  float64
 11  type          18249 non-null  object 
 12  year          18249 non-null  int64  
 13  region        18249 non-null  object 
dtypes: float64(9), int64(2), object(3)
memory usage: 1.9+ MB
```

In [8]:
```
dataset = dataset.drop('Unnamed: 0', axis=1)  # Drop unnecessary column
```

In [9]:
```
dataset['Date'] = pd.to_datetime(dataset['Date'])
```

In [10]:
```
dataset['month'] = dataset['Date'].dt.month
```

In [11]:
```
dataset.head(2)
```

Out[11]:

	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags
0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0.0
1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0.0

In [12]:
```
dataset.isnull().any()
```

Out[12]:
```
Date            False
AveragePrice    False
Total Volume    False
4046            False
4225            False
4770            False
Total Bags      False
Small Bags      False
Large Bags      False
XLarge Bags     False
type            False
year            False
region          False
month           False
dtype: bool
```

In [13]:
```python
dataset.duplicated().any()
```
Out[13]:
```
False
```

Exploratory Data Analysis (EDA)

There are two categorical variables, type and region.

In [14]:
```python
dataset.describe(include='O')  # Pulls out the objects dtypes attributes anc
```
Out[14]:

	type	region
count	18249	18249
unique	2	54
top	conventional	Indianapolis
freq	9126	338

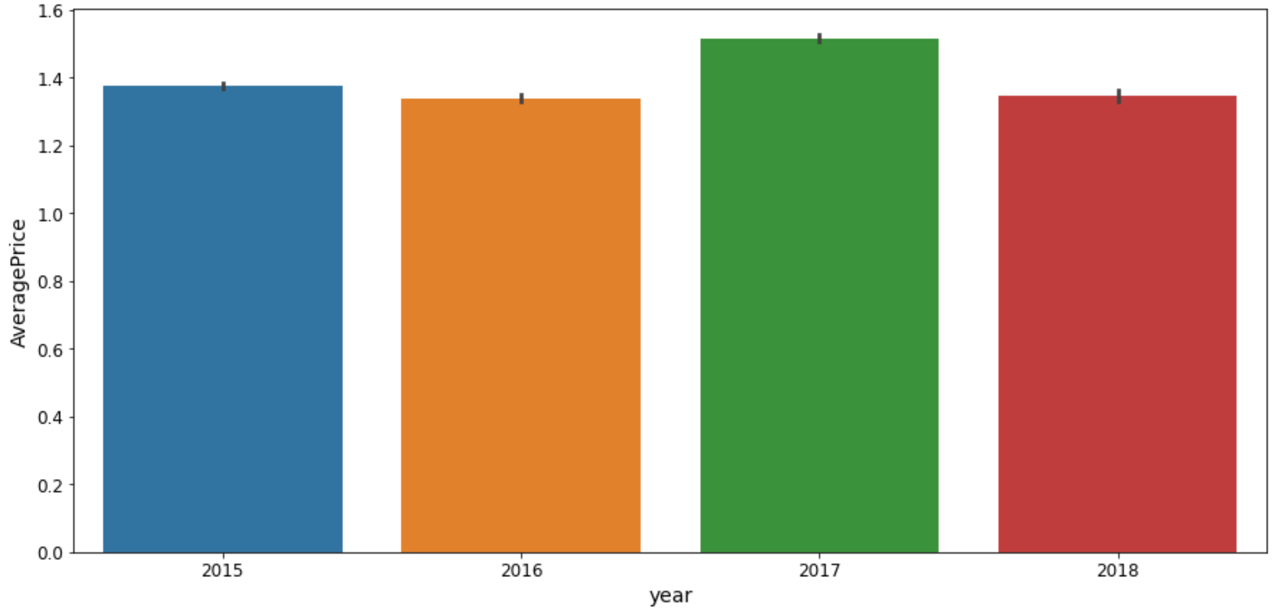
In [15]:
```python
dataset.groupby('year')['type'].value_counts()
```
Out[15]:
```
year  type
2015  conventional    2808
      organic         2807
2016  conventional    2808
      organic         2808
2017  conventional    2862
      organic         2860
2018  conventional     648
      organic          648
Name: type, dtype: int64
```

We can see the total count of both types of avocados for each year. There are almost the same amount of observations for both types in the data set.

In [16]:
```python
dataset.groupby('year')['AveragePrice'].mean()
```
Out[16]:
```
year
2015    1.375590
2016    1.338640
2017    1.515128
2018    1.347531
Name: AveragePrice, dtype: float64
```

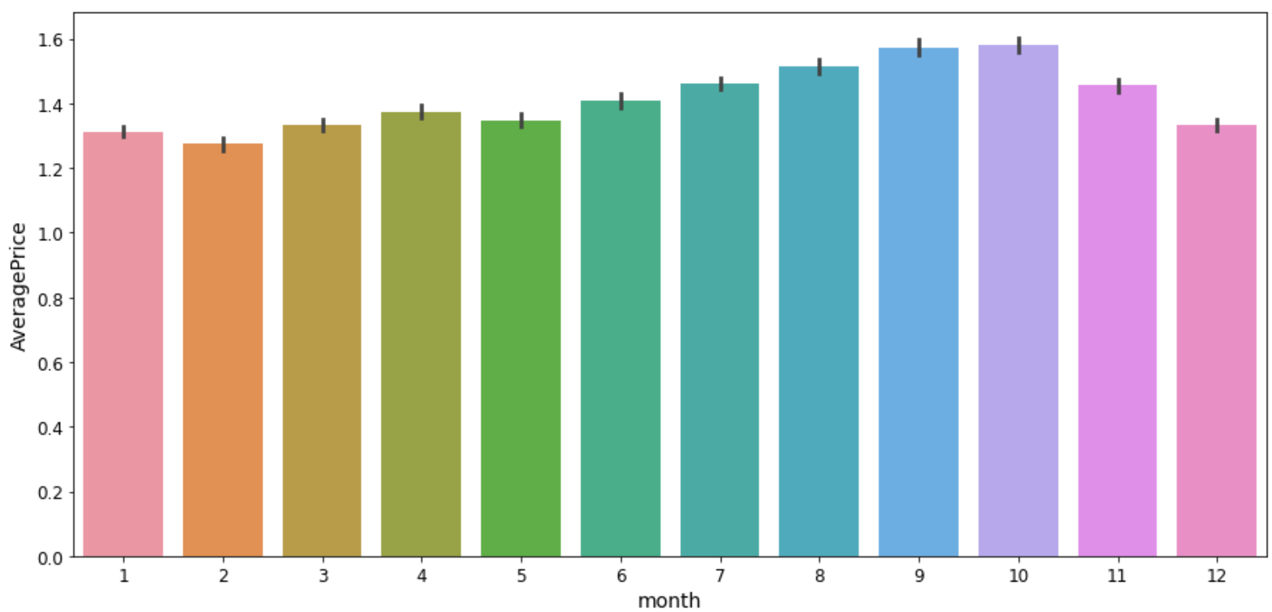
Avocado prices were highest in 2017, followed by 2015. A few plots to see the above observations visually:

In [17]:
```python
sns.barplot(x='year', y='AveragePrice', data=dataset)
```
Out[17]:
```
<AxesSubplot:xlabel='year', ylabel='AveragePrice'>
```



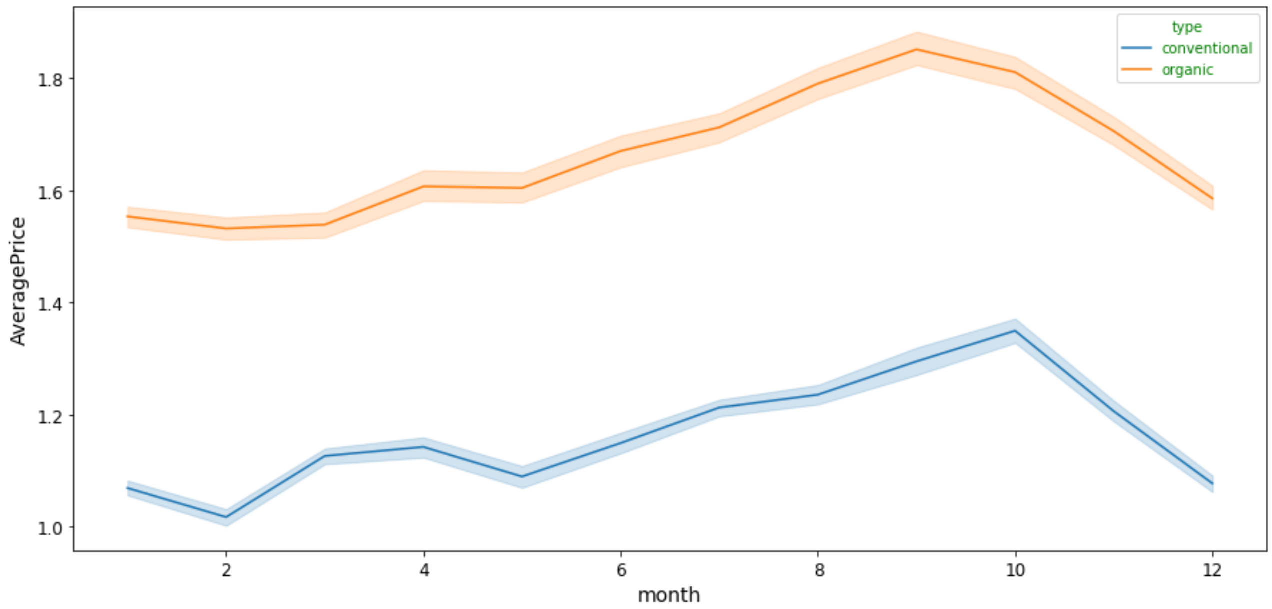
In [18]:
```
sns.barplot(x='month', y='AveragePrice', data=dataset)
```

Out[18]:
```
<AxesSubplot:xlabel='month', ylabel='AveragePrice'>
```



In [19]:
```
sns.lineplot(x='month', y='AveragePrice', hue='type', data=dataset)
```

Out[19]:
```
<AxesSubplot:xlabel='month', ylabel='AveragePrice'>
```



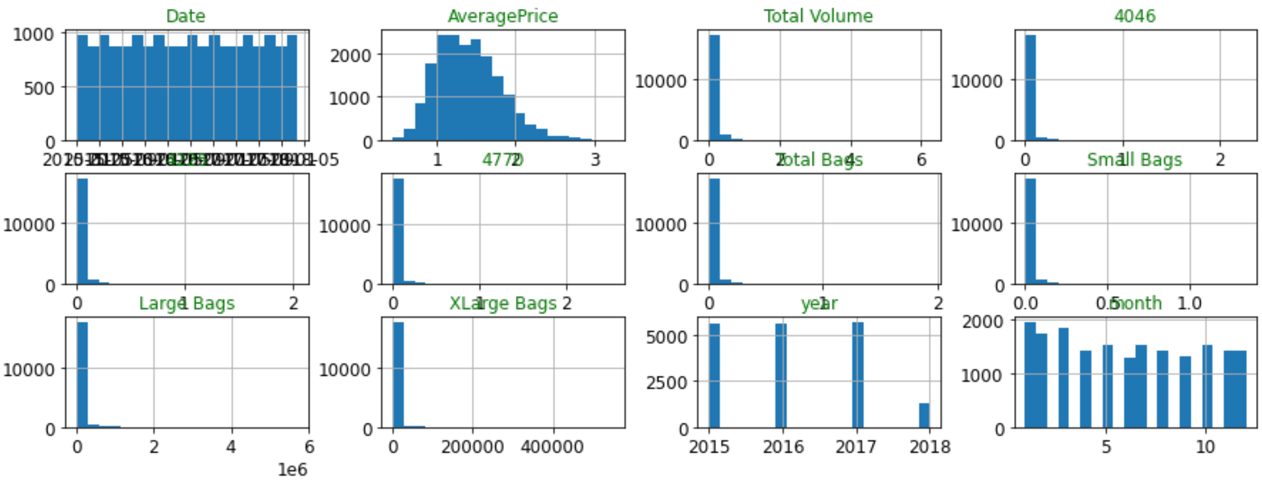
- There is price hike between month 8–10 for both conventional and organic types of avocados.
- From the line plot we can see that the spread in price for organic avocados is greater than that of conventional avocados.

Plotting a Histogram:

- Mainly done to check skewness of variables and potential for outliers.

In [20]:
```
dataset.hist(grid=True,layout=(4,4),bins=20)
```

Out[20]:
```
array([[<AxesSubplot:title={'center':'Date'}>,
        <AxesSubplot:title={'center':'AveragePrice'}>,
        <AxesSubplot:title={'center':'Total Volume'}>,
        <AxesSubplot:title={'center':'4046'}>],
       [<AxesSubplot:title={'center':'4225'}>,
        <AxesSubplot:title={'center':'4770'}>,
        <AxesSubplot:title={'center':'Total Bags'}>,
        <AxesSubplot:title={'center':'Small Bags'}>],
       [<AxesSubplot:title={'center':'Large Bags'}>,
        <AxesSubplot:title={'center':'XLarge Bags'}>,
        <AxesSubplot:title={'center':'year'}>,
        <AxesSubplot:title={'center':'month'}>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]],
      dtype=object)
```



For plotting these histograms, the bin size as 20 gave decent visualizations.

- Average price column is approximately normally distributed over the histogram.
- Rest of the variables don't vary much, so they are almost all left skewed.
- To make the columns normally distributed we will use numPy log to make the skew values as normally distributed.

In [21]:
```
dataset.skew()
```

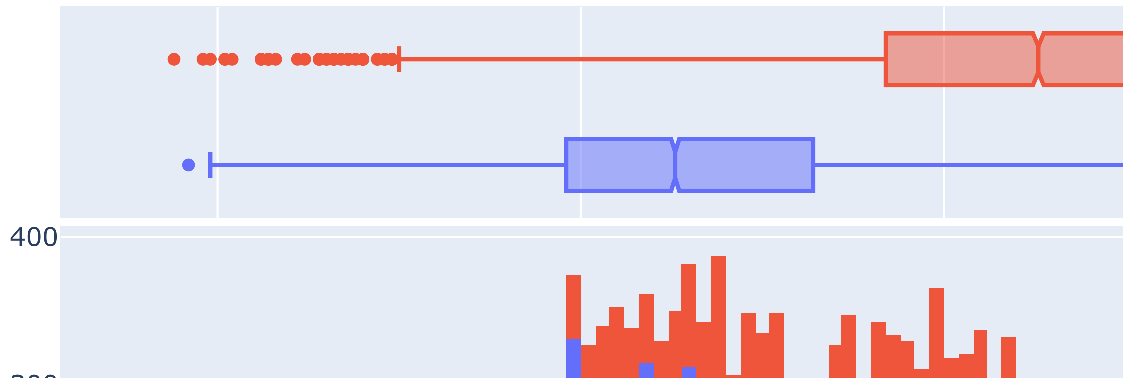
Out[21]:
```
AveragePrice      0.580303
Total Volume      9.007687
4046              8.648220
4225              8.942466
4770             10.159396
Total Bags        9.756072
Small Bags        9.540660
Large Bags        9.796455
XLarge Bags      13.139751
year              0.215339
month             0.106617
dtype: float64
```

In [22]:
```
skew = ('Total Volume', '4046', '4225', '4770', 'Total Bags', 'Small Bags',
for col in skew:
    if dataset.skew().loc[col]>0.55:
        dataset[col] = np.log1p(dataset[col])
```

The best skew value for normally distributed data is very close to zero, so we are using the “log1p” method to make the skew value near to zero.

In [23]:
```
# Let's see the price distribution of two types of avocados.
fig = px.histogram(dataset, x='AveragePrice', color='type',
                   marginal='box',
                   hover_data=dataset.columns)


fig.show()
```



So, on average, organic avocados are more expensive than conventional.

Correlation Matrix:

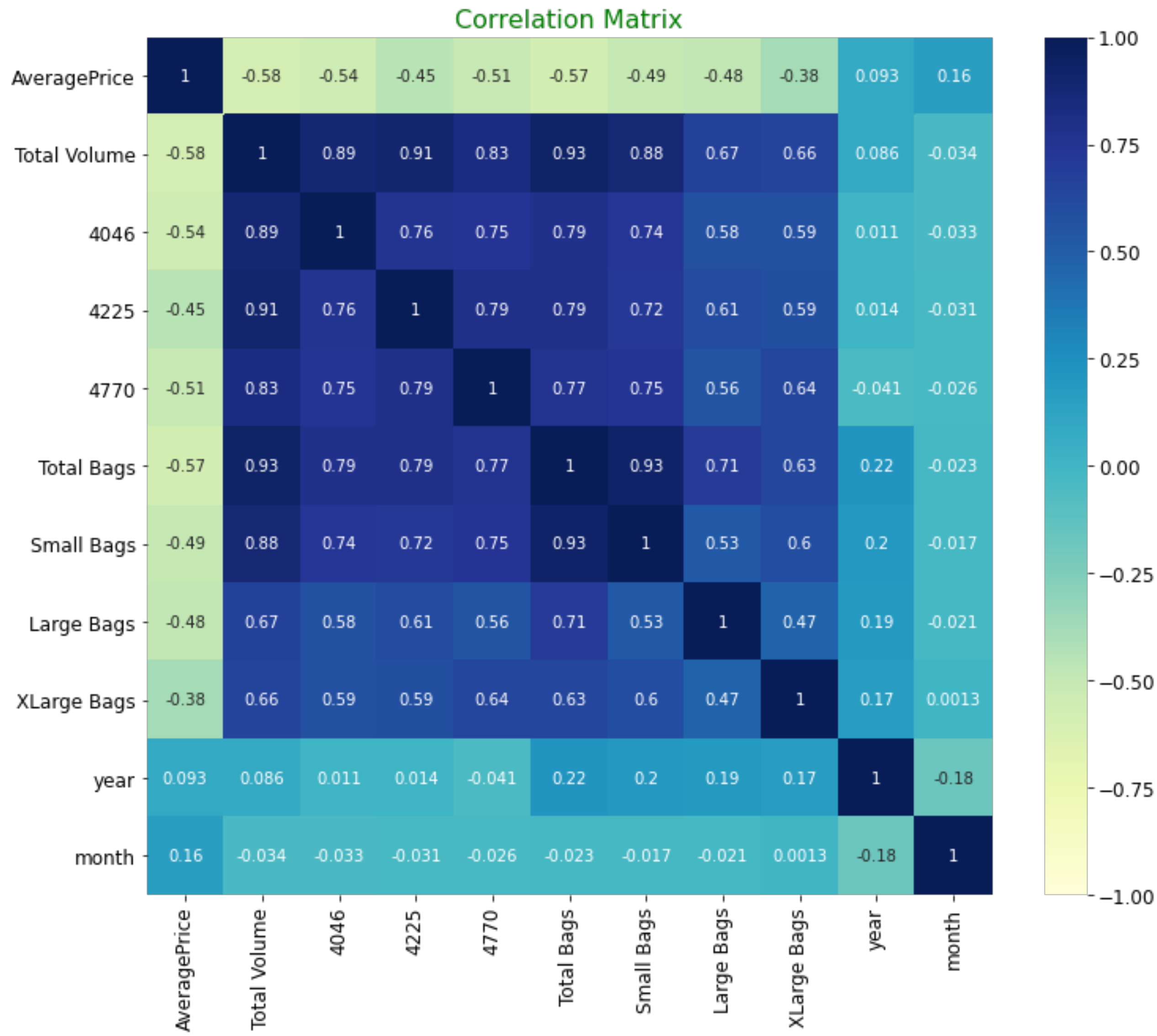
Will use seaborn heatmap to plot the correlated matrix and plot the corr values in the heatmap graph.

In [24]:
```
corr = dataset.corr()
corr

f, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 10))
ax.set_title('Correlation Matrix', fontsize=16)

sns.heatmap(corr, vmin=-1, vmax=1, cmap='YlGnBu', annot=True)
```

Out[24]:
```
<AxesSubplot:title={'center':'Correlation Matrix'}>
```



Feature Engineering

In [25]:
```
# df = dataset
df = dataset.copy()
```

In [26]:
```
# Extracting month
df['month'] = df['Date'].dt.month
```

In [27]:
```
#Introducing a new feature 'season'
df['season'] = df['month']%12 // 3 + 1
# Months will then be:
# Dec, Jan, Feb = 1 (Winter)
# Mar, Apr, May, = 2 (Spring)
# Jun, Jul, Aug = 3 (Summer)
# Sep, Oct, Nov = 4 (Fall)
```

In [28]:
```
# Another possible method but more lines of code:
# conditions = [(df['month'].between(3,5,inclusive=True)),
#             (df['month'].between(6,8,inclusive=True)),
#             (df['month'].between(9,11,inclusive=True)),
#             (df['month'].between(12,2,inclusive=True))]
#
# values = [0,1,2,3]

# df['season'] = np.select(conditions, values)
```

In [29]:
```
# Variables which provide the "same"/"duplicate" information:
# Total Volume = 4046 + 4225 + 4770 + Total Bags
# Total Bags = Small Bags + Large Bags + XLarge Bags
```

In [30]:
```
df.columns
```

Out[30]:
```
Index(['Date', 'AveragePrice', 'Total Volume', '4046', '4225', '4770',
       'Total Bags', 'Small Bags', 'Large Bags', 'XLarge Bags', 'type', 'yea
r',
       'region', 'month', 'season'],
      dtype='object')
```

Train-test splitting

Brief description: The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model. The dataset is large enough to perform the split and since we are not doing classification, balancing of classes shouldn't be an issue.

In [31]:
```
## Separating out different columns into various categories
target_var = ['AveragePrice']
cols_to_remove = ['AveragePrice','Date','4046','4225','4770','Small Bags','L
num_cols = ['Total Volume', 'Total Bags', 'year', 'month', 'season']
cat_cols = ['type','region']
```

In [32]:
```
## Separating out target variable and removing the non-essential columns
y = df[target_var].values
df.drop(cols_to_remove, axis=1, inplace=True)
```

Will use scikit-learn for the implementation of the train-test split evaluation procedure via the train_test_split() function.

In [33]:
```
from sklearn.model_selection import train_test_split
```

In [34]:
```
## Splitting into train and test set
df_train, df_test, y_train, y_test = train_test_split(df, y.ravel(), test_si
```

In [35]:
```
df_train.shape, df_test.shape, y_train.shape, y_test.shape
```

Out[35]:
```
((12226, 7), (6023, 7), (12226,), (6023,))
```

In [36]:
```python
np.mean(y_train), np.mean(y_test)
```

Out[36]:
```
(1.4068673319155898, 1.4041739996679397)
```

Converting categorical columns to numerical columns

There are many ways to convert categorical values into numerical values. Each approach has its own trade-offs and impact on the feature set. We will focus on 2 main methods: Label-Encoding and One-Hot-Encoding. Both of these encoders are part of scikit-learn library and are used to convert text or categorical data into numerical data which the model expects and performs better with.

Label Encoding approach:

This approach is very simple and it involves converting each value in a column to a number.

Category Codes approach (non-sklearn):

This approach requires the category column to be of 'category' datatype. By default, a non-numerical column is of 'object' type. So we need to change type to 'category' before using this approach.

In [37]:
```python
df_train['type_cat'] = df_train.type.astype('category').cat.codes
```

In [38]:
```python
df_train.sample(4)
```

Out[38]:

	Total Volume	Total Bags	type	year	region	month	season	type_cat
4042	11.423663	10.399534	conventional	2016	Louisville	4	2	0
3906	12.466695	11.568266	conventional	2016	LasVegas	11	4	0
14220	10.117346	9.354055	organic	2016	Seattle	1	1	1
5860	13.257633	11.714122	conventional	2017	Boston	5	2	0

In [39]:
```python
df_train.drop('type_cat', axis=1, inplace = True)
```

In [40]:
```python
# The sklearn method
from sklearn.preprocessing import LabelEncoder
```

In [41]:
```python
le = LabelEncoder()
```

In [42]:
```python
# Label encoding of Type variable
df_train['type'] = le.fit_transform(df_train['type'])
```

In [43]:
```
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
le_name_mapping
```

Out[43]:
```
{'conventional': 0, 'organic': 1}
```

In [44]:
```
# What if type column has new values in test set? Need to double-check
le.transform([['organic']])
# le.transform([['ABC']])
```

Out[44]:
```
array([1])
```

In [45]:
```
pd.Series(['ABC']).map(le_name_mapping)
```

Out[45]:
```
0    NaN
dtype: float64
```

In [46]:
```
# Encoding type feature for test set
df_test['type'] = df_test.type.map(le_name_mapping)

# Filling missing/NaN values created due to new categorical levels
df_test['type'].fillna(-1, inplace=True)
```

In [47]:
```
df_train.type.unique(), df_test.type.unique()
```

Out[47]:
```
(array([0, 1]), array([0, 1]))
```

Though label encoding is straightforward, it has one important caveat. Depending on the data values and type of data, label encoding creates a new problem of number sequencing. The problem using the number is that they introduce relation/comparison between them. Apparently, there is no relation between various regions, but when looking at the number, one might think that the 'DallasFtWorth' region has higher precedence over the 'NewYork' region. The algorithm might misunderstand that the data has some kind of hierarchy/order 0 < 1 < 2 ... and might give more weight to 'DallasFtWorth' in calculation than the 'NewYork' region type.

We can address this issue in another common alternative approach called 'One-Hot Encoding'.

One-Hot encoding for categorical variables with multiple levels

In this method, each category value is converted into a new column and assigned a 1 or 0 (notation for true/false) value to the column.

In [48]:
```
# The non-sklearn method
t = pd.get_dummies(df_train, prefix_sep = "_", columns = ['region'])
t.head()
```

Out[48]:

	Total Volume	Total Bags	type	year	month	season	region_Albany	region_Atlanta	reg
4602	11.615241	10.789223	0	2016	6	3	0	0	
10571	7.991535	7.861084	1	2015	3	2	0	0	
18050	9.563945	9.022053	1	2018	2	1	0	0	
15847	8.365796	8.132445	1	2017	2	1	0	0	
8326	17.291699	16.158479	0	2017	11	4	0	0	

5 rows × 60 columns

In [49]:

```
# The sklearn method
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

The OneHotEncoder from the SciKit library only takes numerical categorical values, hence any value of string type should be label encoded before one hot encoded.

In [50]:

```
le_ohe = LabelEncoder()
ohe = OneHotEncoder(handle_unknown = 'ignore', sparse=False)
```

In [51]:

```
enc_train = le_ohe.fit_transform(df_train.region).reshape(df_train.shape[0],
enc_train.shape
```

Out[51]:

```
(12226, 1)
```

In [52]:

```
np.unique(enc_train)
```

Out[52]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53])
```

In [53]:

```
ohe_train = ohe.fit_transform(enc_train)
ohe_train
```

Out[53]:

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

In [54]:

```
le_ohe_name_mapping = dict(zip(le_ohe.classes_, le_ohe.transform(le_ohe.clas
le_ohe_name_mapping
```

```
Out[54]: {'Albany': 0,
 'Atlanta': 1,
 'BaltimoreWashington': 2,
 'Boise': 3,
 'Boston': 4,
 'BuffaloRochester': 5,
 'California': 6,
 'Charlotte': 7,
 'Chicago': 8,
 'CincinnatiDayton': 9,
 'Columbus': 10,
 'DallasFtWorth': 11,
 'Denver': 12,
 'Detroit': 13,
 'GrandRapids': 14,
 'GreatLakes': 15,
 'HarrisburgScranton': 16,
 'HartfordSpringfield': 17,
 'Houston': 18,
 'Indianapolis': 19,
 'Jacksonville': 20,
 'LasVegas': 21,
 'LosAngeles': 22,
 'Louisville': 23,
 'MiamiFtLauderdale': 24,
 'Midsouth': 25,
 'Nashville': 26,
 'NewOrleansMobile': 27,
 'NewYork': 28,
 'Northeast': 29,
 'NorthernNewEngland': 30,
 'Orlando': 31,
 'Philadelphia': 32,
 'PhoenixTucson': 33,
 'Pittsburgh': 34,
 'Plains': 35,
 'Portland': 36,
 'RaleighGreensboro': 37,
 'RichmondNorfolk': 38,
 'Roanoke': 39,
 'Sacramento': 40,
 'SanDiego': 41,
 'SanFrancisco': 42,
 'Seattle': 43,
 'SouthCarolina': 44,
 'SouthCentral': 45,
 'Southeast': 46,
 'Spokane': 47,
 'StLouis': 48,
 'Syracuse': 49,
 'Tampa': 50,
 'TotalUS': 51,
 'West': 52,
 'WestTexNewMexico': 53}
```

In [55]:
```
# Encoding Region feature for test set
enc_test = df_test.region.map(le_ohe_name_mapping).ravel().reshape(-1,1)

# Filling missing/NaN values created due to new categorical levels
enc_test[np.isnan(enc_test)] = 9999
```

In [56]:
```
np.unique(enc_test)
```

Out[56]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53])
```

In [57]:
```
ohe_test = ohe.transform(enc_test)
```

In [58]:
```
### Show what happens when a new value is inputted into the OHE, basically b
ohe.transform(np.array([[9999]]))
```

Out[58]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0.]])
```

Adding the one-hot encoded columns to the dataframe and removing the original feature

In [59]:
```
cols = ['region_' + str(x) for x in le_ohe_name_mapping.keys()]
cols
```

Out[59]:
```
['region_Albany',
 'region_Atlanta',
 'region_BaltimoreWashington',
 'region_Boise',
 'region_Boston',
 'region_BuffaloRochester',
 'region_California',
 'region_Charlotte',
 'region_Chicago',
 'region_CincinnatiDayton',
 'region_Columbus',
 'region_DallasFtWorth',
 'region_Denver',
 'region_Detroit',
 'region_GrandRapids',
 'region_GreatLakes',
 'region_HarrisburgScranton',
 'region_HartfordSpringfield',
 'region_Houston',
 'region_Indianapolis',
 'region_Jacksonville',
 'region_LasVegas',
 'region_LosAngeles',
 'region_Louisville',
 'region_MiamiFtLauderdale',
 'region_Midsouth',
 'region_Nashville',
 'region_NewOrleansMobile',
 'region_NewYork',
 'region_Northeast',
 'region_NorthernNewEngland',
 'region_Orlando',
 'region_Philadelphia',
 'region_PhoenixTucson',
 'region_Pittsburgh',
 'region_Plains',
 'region_Portland',
 'region_RaleighGreensboro',
 'region_RichmondNorfolk',
 'region_Roanoke',
 'region_Sacramento',
 'region_SanDiego',
 'region_SanFrancisco',
 'region_Seattle',
 'region_SouthCarolina',
 'region_SouthCentral',
 'region_Southeast',
 'region_Spokane',
 'region_StLouis',
 'region_Syracuse',
 'region_Tampa',
 'region_TotalUS',
 'region_West',
 'region_WestTexNewMexico']
```

In [60]:
```
## Adding to the respective dataframes
df_train = pd.concat([df_train.reset_index(), pd.DataFrame(ohe_train, column
df_test = pd.concat([df_test.reset_index(), pd.DataFrame(ohe_test, columns =
```

In [61]:
```
## Drop the region column
df_train.drop(['region'], axis = 1, inplace=True)
df_test.drop(['region'], axis = 1, inplace=True)
```

In [62]:
```
df_train.head()
```

Out[62]:

	Total Volume	Total Bags	type	year	month	season	region_Albany	region_Atlanta	region_B
0	11.615241	10.789223	0	2016	6	3	0.0	0.0	
1	7.991535	7.861084	1	2015	3	2	0.0	0.0	
2	9.563945	9.022053	1	2018	2	1	0.0	0.0	
3	8.365796	8.132445	1	2017	2	1	0.0	0.0	
4	17.291699	16.158479	0	2017	11	4	0.0	0.0	

5 rows × 60 columns

Though this approach eliminates the hierarchy/order issues it does have the downside of adding more columns to the data set. This can cause the number of columns to expand greatly if you have many unique values in a category column. In this case it was manageable, but it will get really challenging to manage when encoding gives many columns.

Quick Regression Modeling (Linear Regression & Random Forest)

In [63]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor

from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_erro
```

Pipeline sequentially applies a list of transforms and a final estimator. The intermediate steps of pipeline must implement fit and transform methods and the final estimator only needs to implement fit.

As the name suggests, the pipeline class allows sticking multiple processes into a single scikit-learn estimator. The pipeline class has fit, predict and score method just like any other estimator in scikit-learn (ex. LinearRegression). Pipeline is necessary at times as it helps to enforce desired order of steps, not only creating a convenient work-flow, but also ensures the reproducibility of the work.

Will use StandardScaler, which subtracts the mean from each features and then scales to unit variance.

In [64]:
```
pipe0 = Pipeline([('scaler', StandardScaler()), ('lr', LinearRegression())])
pipe0.fit(df_train, y_train)
y_pred0 = pipe0.predict(df_test)
print("R2: {}".format(r2_score(y_test, y_pred0)))
print("MAE: {}".format(mean_absolute_error(y_test, y_pred0)))
```

```
R2: 0.7264588755025747
MAE: 0.15731023867447108
```

In [65]:
```
pipe = Pipeline([('scaler', StandardScaler()), ('rf', RandomForestRegressor(
pipe.fit(df_train, y_train)
y_pred = pipe.predict(df_test)
print("R2: {}".format(r2_score(y_test, y_pred)))
print("MAE: {}".format(mean_absolute_error(y_test, y_pred)))
```

```
R2: 0.9013583950633716
MAE: 0.08860094637223973
```

In [66]:
```
rf = RandomForestRegressor()
rf.fit(df_train, y_train)
len(rf.estimators_)
```

Out[66]:
```
100
```

In [67]:
```
rf.estimators_[0].tree_.max_depth
```

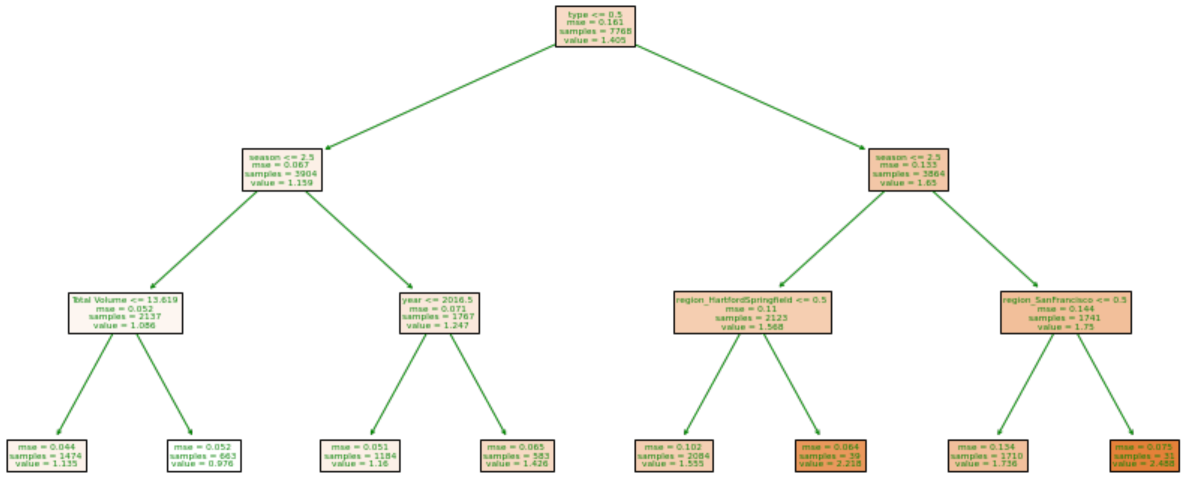
Out[67]:
```
50
```

In [68]:
```
rf = RandomForestRegressor(n_estimators=100, max_depth=3)
rf.fit(df_train, y_train)
```

Out[68]:
```
RandomForestRegressor(max_depth=3)
```

In [69]:
```
from sklearn import tree
```

In [70]:
```
_ = tree.plot_tree(rf.estimators_[0], feature_names=df_train.columns, filled
```



In [71]:
```python
pipe2 = Pipeline([('scaler', StandardScaler()), ('xgb', XGBRegressor())])
pipe2.fit(df_train, y_train)
y_pred2 = pipe2.predict(df_test)
print("R2: {}".format(r2_score(y_test, y_pred2)))
print("MAE: {}".format(mean_absolute_error(y_test, y_pred2)))
```

```
R2: 0.8967389132328858
MAE: 0.09407359496493801
```

In [72]:
```python
pd.DataFrame(pipe2['xgb'].feature_importances_, index=df_train.columns,
             columns=['Feature Importances']).sort_values(by='Feature Import
```

Out[72]:

	Feature Importances
type	0.475892
region_HartfordSpringfield	0.052715
region_NewYork	0.028565
region_SanFrancisco	0.027738
region_DallasFtWorth	0.023907

Time Series Forecasting

Background:

A time series is a sequence of observations recorded at regular time intervals.

Depending on the frequency of observations, a time series may typically be hourly, daily, weekly, monthly, quarterly and annual. Sometimes, you might have seconds and minute-wise time series as well, like stock market data for various stocks at second-level intervals, the number of clicks and user visits every minute, etc.

Forecasting is the step where you want to predict the future values the series is going to take. Forecasting a time series (like price, demand, and sales) is often of tremendous commercial value.

Time Series Components:

A useful abstraction for selecting forecasting methods is to break a time series down into systematic and unsystematic components. A given time series is thought to consist of three systematic components including level, trend, seasonality, and one non-systematic component called noise.

A series is thought to be an aggregate or combination of these four components. All series have a level and noise component. The trend and seasonality components are optional.

It is helpful to think of the components as combining either additively or multiplicatively.

Will focus on a particular type of forecasting method called ARIMA modeling.

ARIMA

ARIMA, short for 'Auto Regressive Integrated Moving Average', is a forecasting algorithm that 'explains' a given time series based on its own past values, that is, its own lags and the lagged forecast errors, so that the equation can be used to forecast future values.

Any 'non-seasonal' time series that exhibits patterns and is not a random white noise can be modeled with ARIMA models.

An ARIMA model is characterized by 3 terms - p, d, q:

- p is the order of the AR term
- d is the number of differencing required to make the time series stationary
- q is the order of the MA term

The first step to build an ARIMA model is to **make the time series stationary**. Why?

Because the term 'Auto Regressive' in ARIMA means it is a linear regression model that uses its own lags as predictors. Linear regression models, as is well known, work best when the predictors are not correlated and are independent of each other.

How do we make a series stationary?

The most common approach is to difference it. That is, subtract the previous value from the current value. Sometimes, depending on the complexity of the series, more than one differencing may be needed.

d is the minimum number of differencing needed to make the series stationary. And if the time series is already stationary, then d = 0.

A pure **Auto Regressive (AR only) model** is one where Y_t depends only on its own lags. And, *p* is the order of the 'Auto Regressive' (AR) term. It refers to the number of lags of Y to be used as predictors.

A pure **Moving Average (MA only) model** is one where Y_t depends only on the lagged forecast errors. And, *q* is the order of the 'Moving Average' (MA) term. It refers to the number of lagged forecast errors that should go into the ARIMA Model.

ARIMA model in words:

Predicted Y_t = Constant + Linear combination of Lags of Y(up to p lags) + Linear Combination of Lagged forecast errors (up to q lags)

Choose parameters for ARIMA

In [73]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [74]:
```
df_ar = pd.read_csv('avocado.csv', index_col=0)
df_ar['Date'] = pd.to_datetime(df_ar['Date'])  # Convert to datetime format
df_ar.set_index('Date', inplace=True)  # Set date as index
df_ar = df_ar[['AveragePrice']]  # Select only "AveragePrice" column
df_ar = df_ar.resample('W').mean() # Get weekly mean values for Average Pric
```

In [75]:
```
df_ar.head()
```

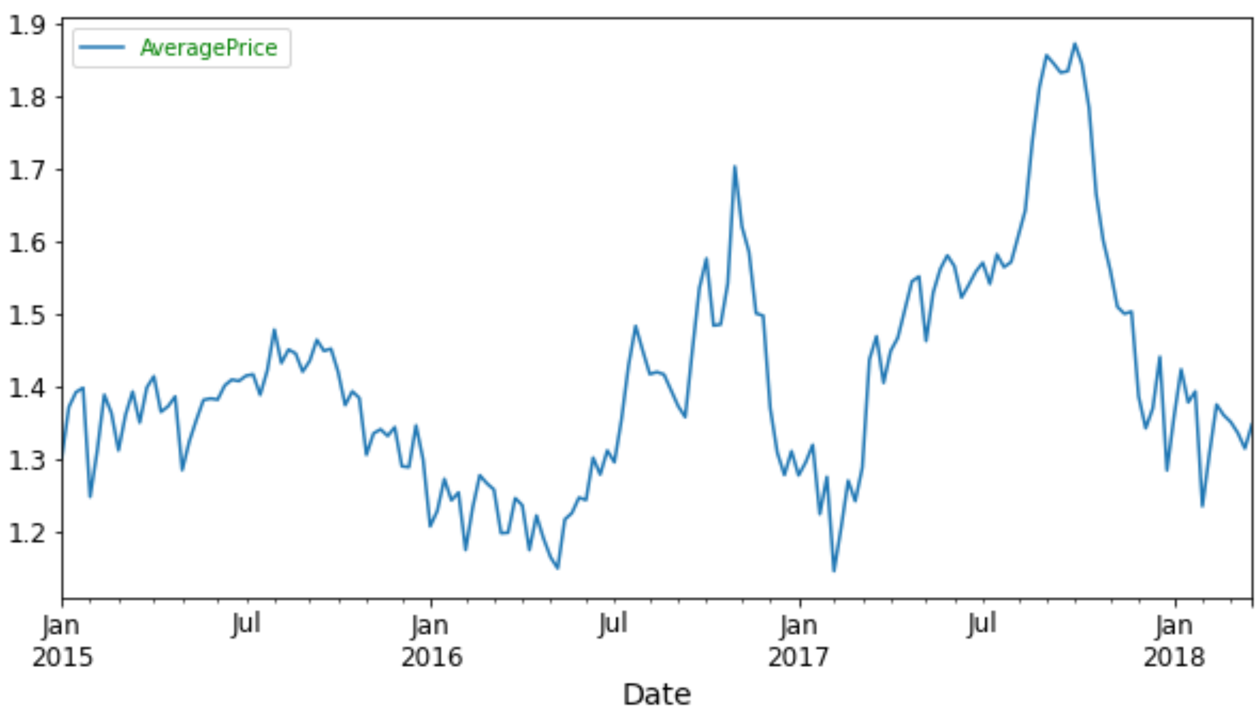
Out [75]:

Date	AveragePrice
2015-01-04	1.301296
2015-01-11	1.370648
2015-01-18	1.391111
2015-01-25	1.397130
2015-02-01	1.247037

In [76]:

```
plt.rcParams['figure.figsize'] = [10, 5]
df_ar.plot()
```

Out[76]: <AxesSubplot:xlabel='Date'>



As we can see from the plot, avocado prices were relatively flat for 2015 and much of 2016. It wasn't until the middle of 2016 that there were "large" swings in prices that were duplicated again in 2017.

Let's look at the components of our time series:

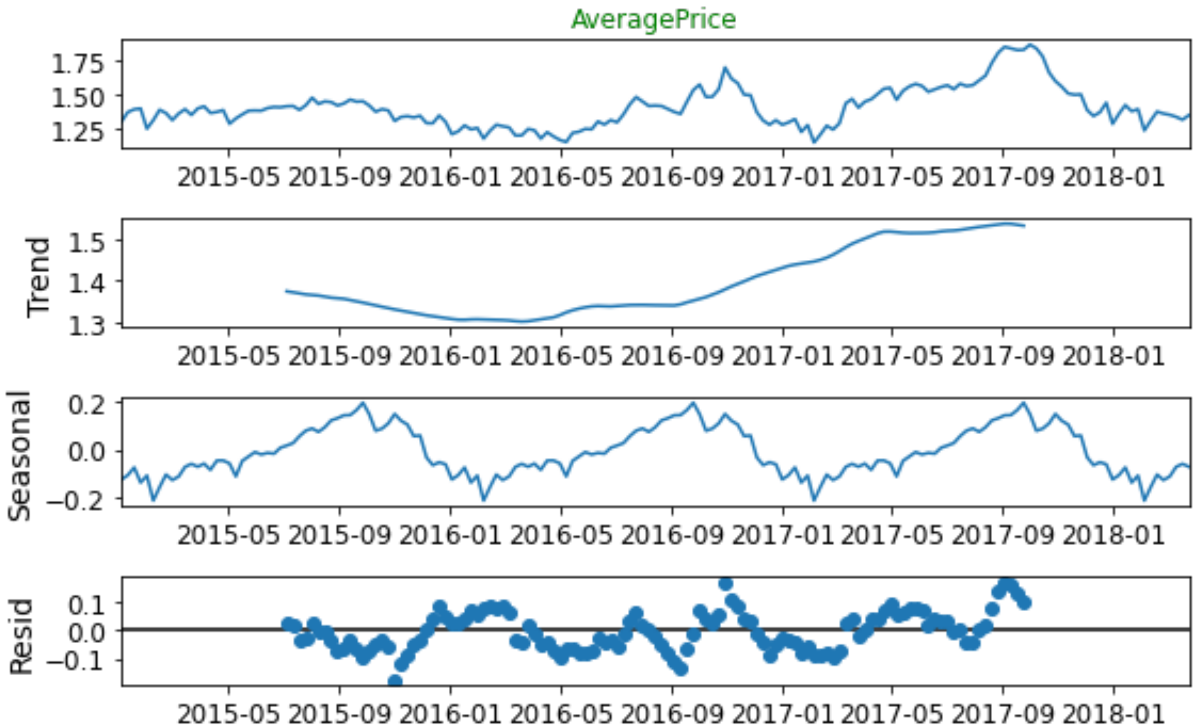
We will use the additive model, which suggests that the components are added together as follows:

> Y_t = Level + Trend + Seasonality + Noise

An additive model is linear where changes over time are consistently made by the same amount.

In [77]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
```

In [78]:
```
plt.rcParams['figure.figsize'] = [8, 5]
a = seasonal_decompose(df_ar["AveragePrice"], model = "add")
a.plot()
plt.ioff()
```



Find order of differencing i.e. 'd':

As stated earlier, the purpose of differencing is to make the time series stationary.

But we need to be careful to not over-difference the series. Because, an over differenced series may still be stationary, which in turn will affect the model parameters.

First, we will check stationarity using the Augmented Dickey Fuller Test. The null hypothesis of the ADF test is that the time series is non-stationary. So, if the p-value of the test is less than the significance level (0.05) then you reject the null hypothesis and infer that the time series is indeed stationary.

In [79]:
```
from statsmodels.tsa.stattools import adfuller
```

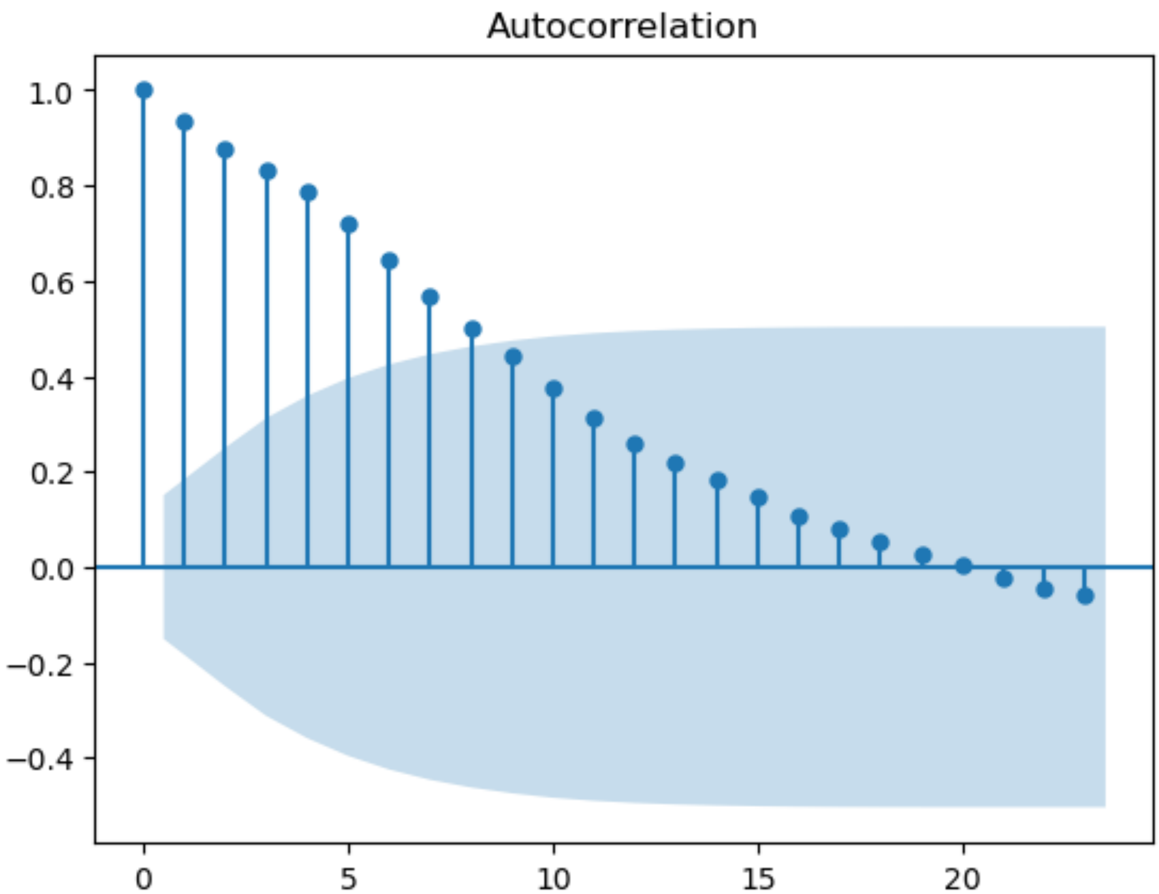
In [80]:
```
result = adfuller(df_ar.AveragePrice.dropna())
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

```
ADF Statistic: -2.357817
p-value: 0.153998
```

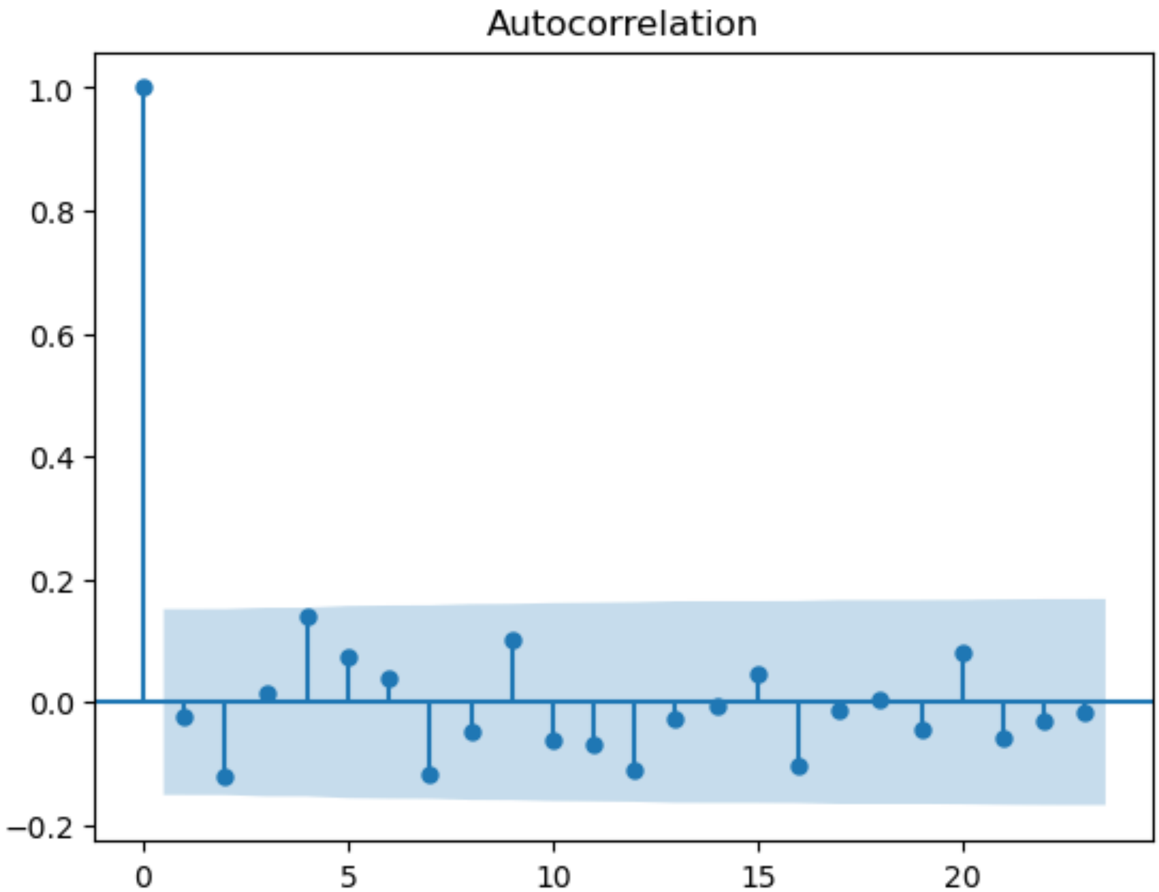
Since p-value is greater than the significance level, let's difference the series and see what the autocorrelation plot looks like.

In [81]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

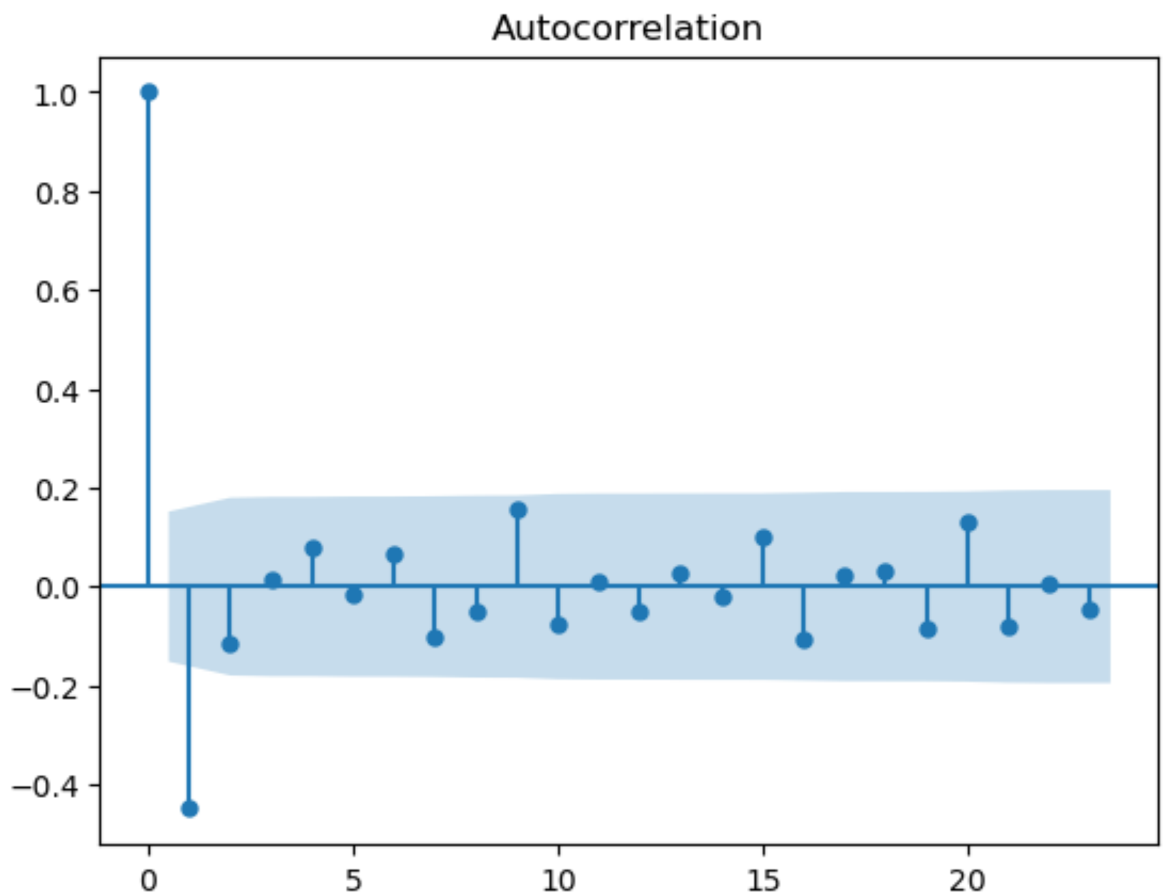
In [82]:
```
plt.style.use('default')
plot_acf(df_ar.AveragePrice);
```



In [83]:
```
# 1st order differencing
plot_acf(df_ar.AveragePrice.diff().dropna());
```



In [84]:
```
# 2nd order differencing
plot_acf(df_ar.AveragePrice.diff().diff().dropna());
```



From above plots we can see that the time series reaches stationarity with one order of differencing.

We can also use 'ndiffs' to estimate 'd'. It performs a test of stationarity for different levels of d to estimate the number of differences required to make a given time series stationary. It will select the maximum value of d for which the time series is judged stationary by the statistical test.

In [85]:
```
## Using pmdarima for this as it very convenient
from pmdarima.arima.utils import ndiffs
```

In [86]:
```
## Adf Test
ndiffs(df_ar.AveragePrice, test='adf')
```

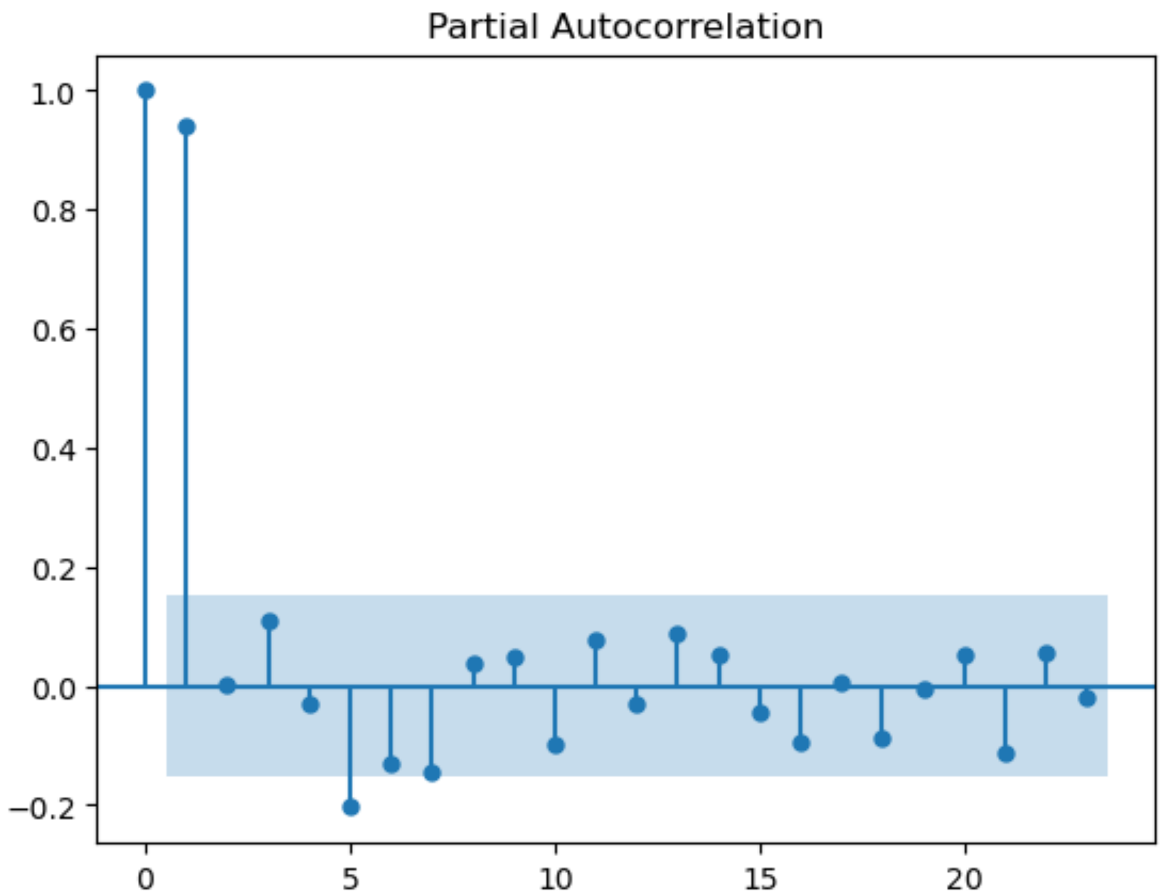
Out[86]:
```
1
```

Find order of AR term i.e. 'p':

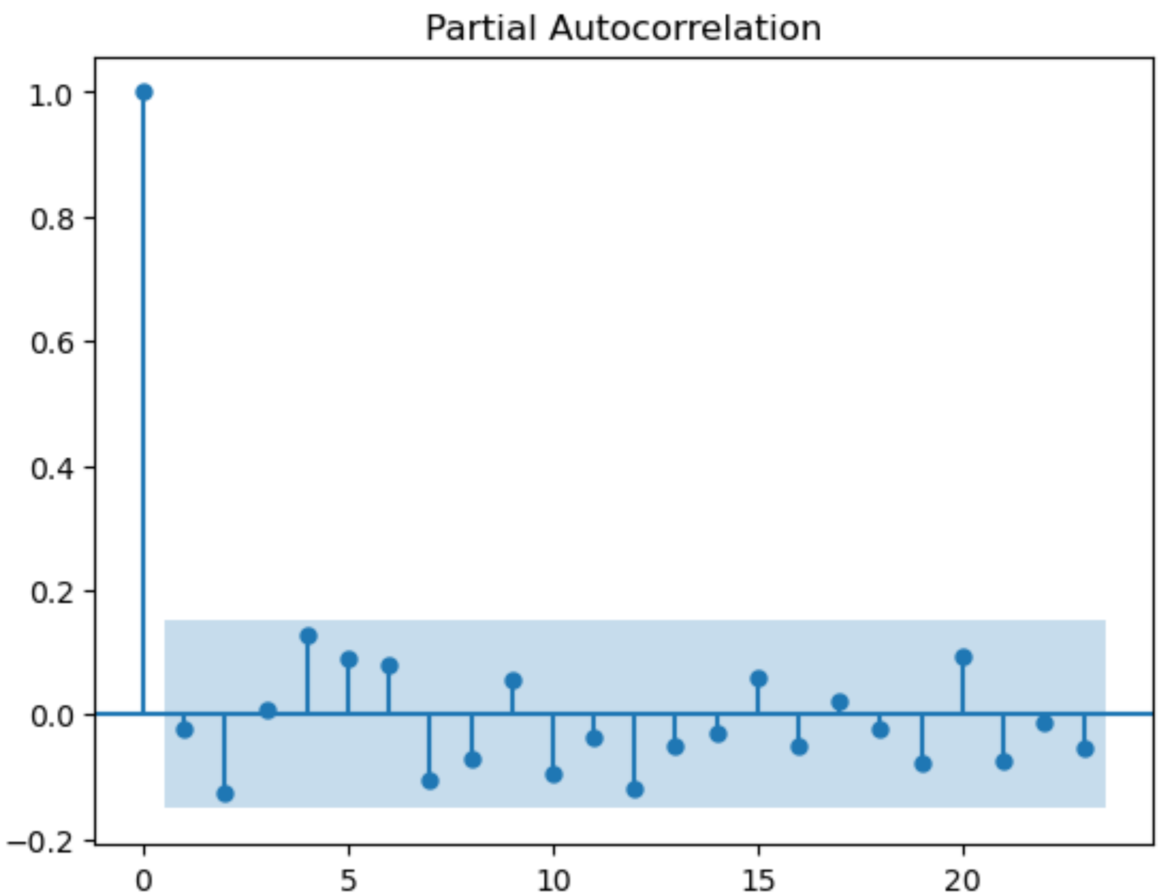
The next step is to identify if the model needs any AR terms. You can find out the required number of AR terms by inspecting the Partial Autocorrelation (PACF) plot. Partial autocorrelation can be thought of as the correlation between the series and its lag, after excluding the contributions from the intermediate lags.

Let's see what partial autocorrelation plots look like:

In [87]:
```
plot_pacf(df_ar.AveragePrice);
```



In [88]:
```
# 1st order differencing
plot_pacf(df_ar.AveragePrice.diff().dropna());
```



We can observe that the PACF without any lag is quite significant and well within the significance limit (blue region).So we will choose p as 0.

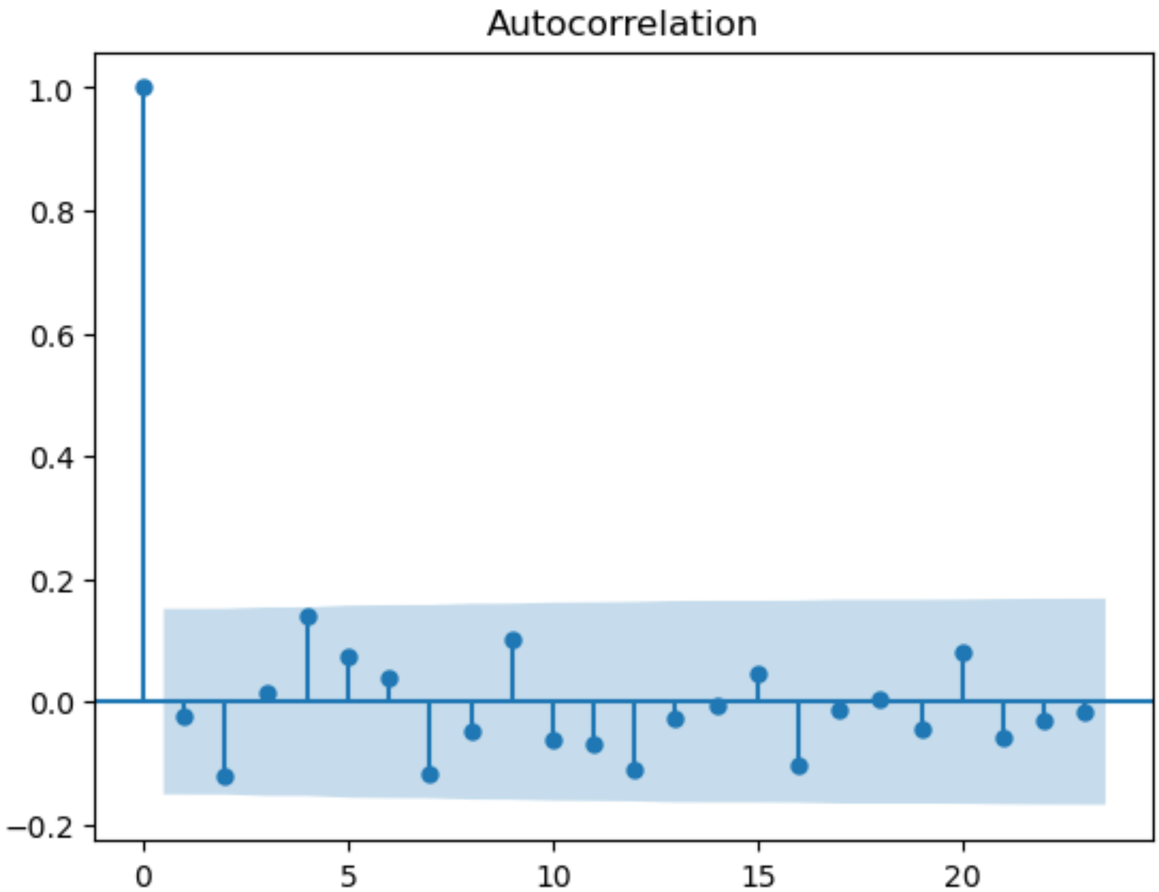
Find order of MA term i.e. 'q':

Just like how we looked at the PACF plot for the number of AR terms, you can look at the ACF plot for the number of MA terms. An MA term is technically, the error of the lagged forecast.

The ACF tells how many MA terms are required to remove any autocorrelation in the stationarized series.

Let’s see the autocorrelation plot of the differenced series:

In [89]:
```
plot_acf(df_ar.AveragePrice.diff().dropna());
```



The lags are well within the significance limit. So, we wil choose q as 0.

Build the ARIMA Model

In [90]:
```
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_errc
import warnings
warnings.filterwarnings('ignore')
```

In [91]:

```
# Evaluate an ARIMA model for a given order (p,d,q)
# This finds the model with the best p, d, q-values that yield the lowest MS
def evaluate_arima_model(X, arima_order):
    # prepare training dataset
    train_size = int(len(X) * 0.66)
    train, test = X[0:train_size], X[train_size:]
    history = [x for x in train]
    # make predictions
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit(disp=0)
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test[t])
    # calculate out of sample error
    error = mean_squared_error(test, predictions)
    return error

# Evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
    dataset = dataset.astype('float32')
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    mse = evaluate_arima_model(dataset, order)
                    if mse < best_score:
                        best_score, best_cfg = mse, order
                    print('ARIMA%s MSE=%.3f' % (order,mse))
                except:
                    continue
    print('Best ARIMA%s MSE=%.3f' % (best_cfg, best_score))
    return best_cfg
```

In [92]:

```
# Evaluate parameters
p_values = range(1, 2)
d_values = range(0, 4)
q_values = range(0, 2)

best_order = evaluate_models(df_ar.values, p_values, d_values, q_values)
```

```
ARIMA(1, 0, 0) MSE=0.004
ARIMA(1, 0, 1) MSE=0.004
ARIMA(1, 1, 0) MSE=0.004
ARIMA(1, 1, 1) MSE=0.004
ARIMA(1, 2, 0) MSE=0.006
Best ARIMA(1, 0, 0) MSE=0.004
```

In [93]:
```
# Instantiate the ARIMA model
model = ARIMA(df_ar['AveragePrice'], order = best_order)

# Fit the model
results_ARIMA = model.fit()

# Collect the predicted results, rounding to two to indicate dollars and cen
predictions = round(results_ARIMA.predict(), 2)

# Put the predictions into a DataFrame with Date and Predicted Price columns
preds = pd.DataFrame(list(zip(list(predictions.index),list(predictions))),cc
'PredictedPrice']).set_index('Date')

# Combine the original data set with the predicted data
predicted_df = pd.merge(df_ar[1:], preds, left_index=True, right_index=True)
```

In [94]:
```
results_ARIMA.summary()
```

Out[94]:

ARMA Model Results

Dep. Variable:	AveragePrice	**No. Observations:**	169
Model:	ARMA(1, 0)	**Log Likelihood**	256.231
Method:	css-mle	**S.D. of innovations**	0.053
Date:	Wed, 20 Jul 2022	**AIC**	-506.461
Time:	00:25:57	**BIC**	-497.072
Sample:	01-04-2015	**HQIC**	-502.651
	- 03-25-2018		

	coef	std err	z	P>|z|	[0.025	0.975]
const	1.3943	0.057	24.556	0.000	1.283	1.506
ar.L1.AveragePrice	0.9336	0.026	36.108	0.000	0.883	0.984

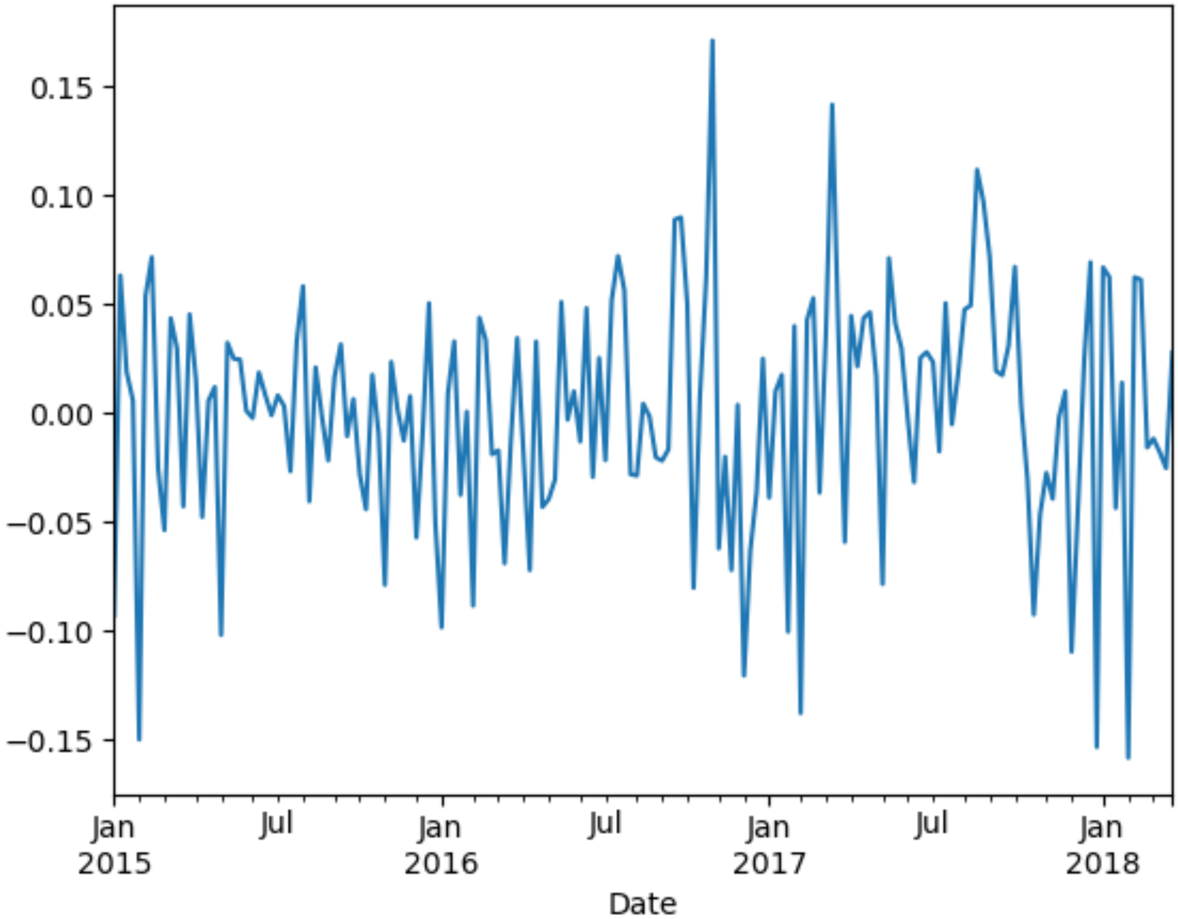
Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.0711	+0.0000j	1.0711	0.0000

Let’s plot the residuals to ensure there are no patterns (that is, look for constant mean and variance).

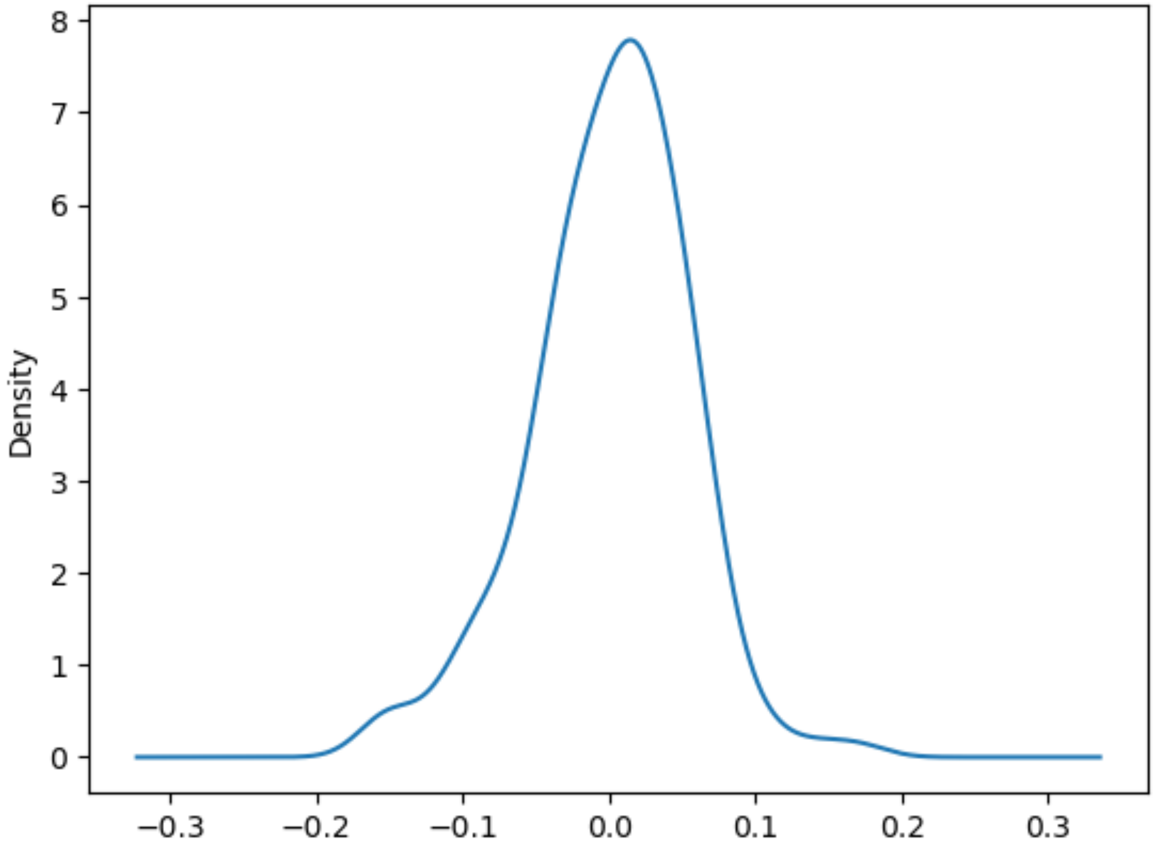
In [95]:
```
# line plot of residuals
residuals = results_ARIMA.resid
residuals.plot()
```

Out[95]:
```
<AxesSubplot:xlabel='Date'>
```



In [96]:
```
# density plot of residuals
residuals.plot(kind='kde')
```

Out[96]:
```
<AxesSubplot:ylabel='Density'>
```



The residual errors seem fine with near zero mean and uniform variance.

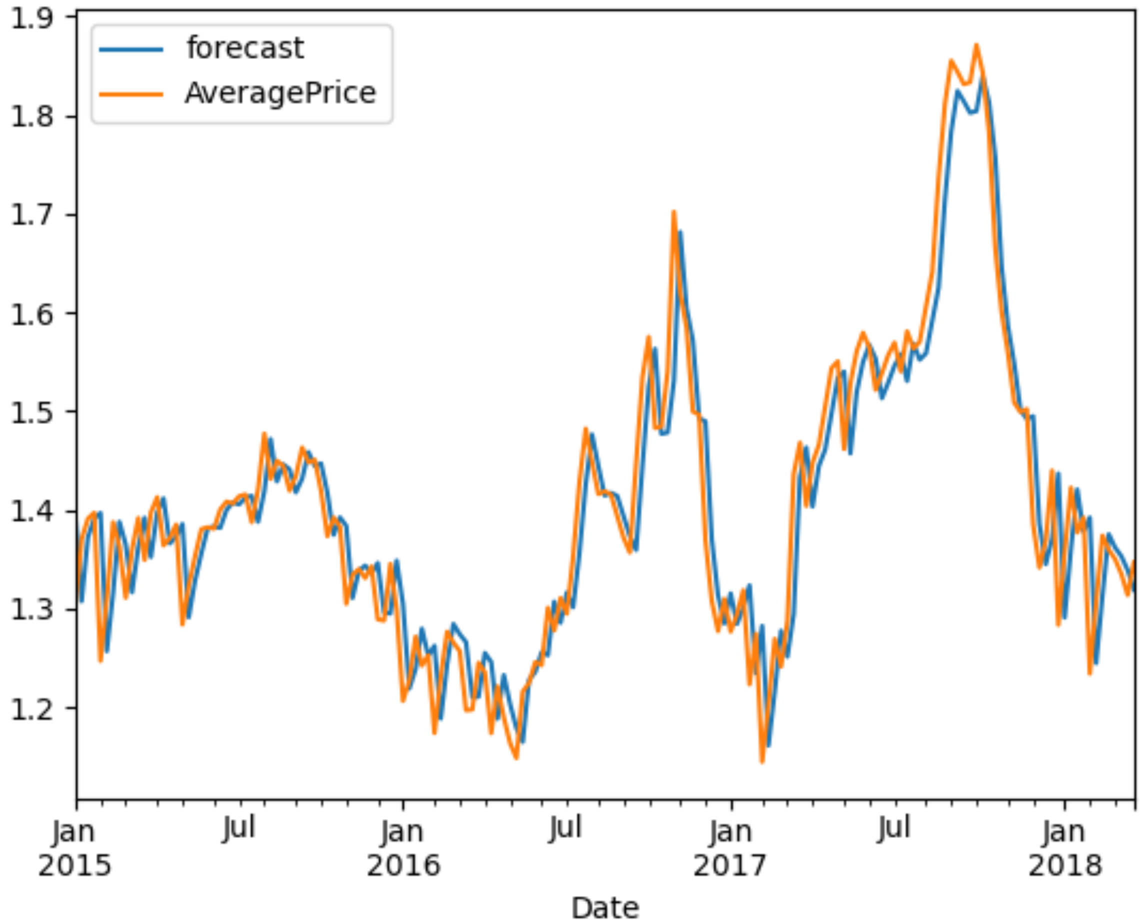
In [97]:
```
print(residuals.describe())
```

```
count    169.000000
mean       0.000514
std        0.053380
min       -0.158286
25%       -0.028726
50%        0.005806
75%        0.033593
max        0.171022
dtype: float64
```

Forecasting and Evaluation

Let's plot the actuals against the fitted values using plot_predict(). When "dynamic" is set as 'False', the in-sample lagged values are used for prediction.

In [98]:
```
results_ARIMA.plot_predict(dynamic=False)
plt.ioff()
```

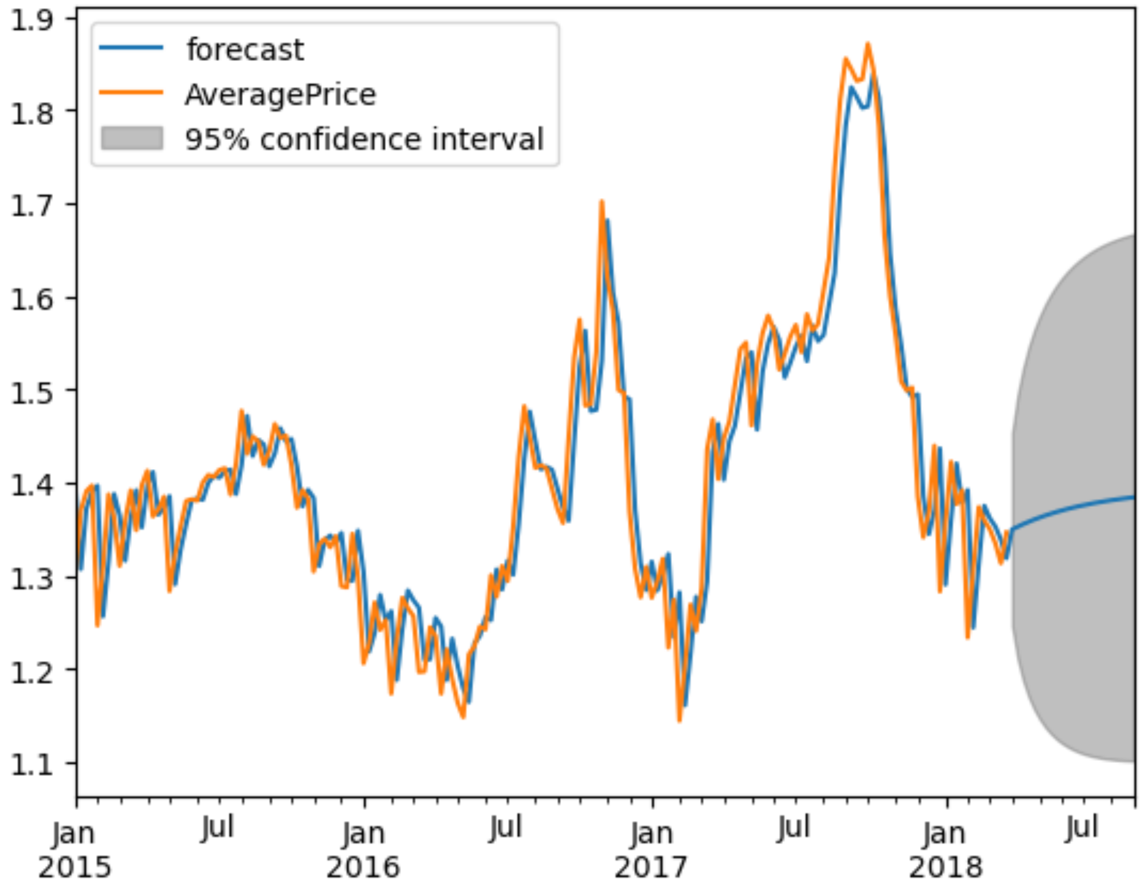


In [99]:
```
print("\tMean Absolute Percentage Error:",
      np.mean(np.abs(predicted_df['PredictedPrice'] - predicted_df['AverageP
print("\tMean Absolute Error:",
      mean_absolute_error(predicted_df['AveragePrice'], predicted_df['Predic
print("\tMean Squared Error:",
      mean_squared_error(predicted_df['AveragePrice'], predicted_df['Predict
print("\tRoot Mean Squared Error:",
      np.sqrt(mean_squared_error(predicted_df['AveragePrice'],predicted_df['
print("\tR2 Score:",
      r2_score(predicted_df['AveragePrice'], predicted_df['PredictedPrice']))
```

```
        Mean Absolute Percentage Error: 0.029476017003040138
        Mean Absolute Error: 0.04092957791458571
        Mean Squared Error: 0.0028161287906848203
        Root Mean Squared Error: 0.05306721012720398
        R2 Score: 0.8772991325397963
```

Around a 2.9% MAPE implies that the model is 97.1% accurate in making predictions.

In [100…
```
results_ARIMA.plot_predict(end='2018-08-31')
plt.ioff()
```



The problem with a plain ARIMA model is it does not support seasonality. We turn to SARIMA.

SARIMA

If the time series has defined seasonality, then SARIMA, which uses seasonal differencing, is a more appropriate model.

Seasonal differencing is similar to regular differencing, but instead of subtracting consecutive terms, you subtract the value from previous season.

So the model will be represented as SARIMA(p,d,q)x(P,D,Q), where, P, D and Q are SAR, order of seasonal differencing and SMA terms respectively and 'x' is the frequency of the time series.

If the model has well defined seasonal patterns, then we enforce D=1 for a given frequency 'x'.

In [101…

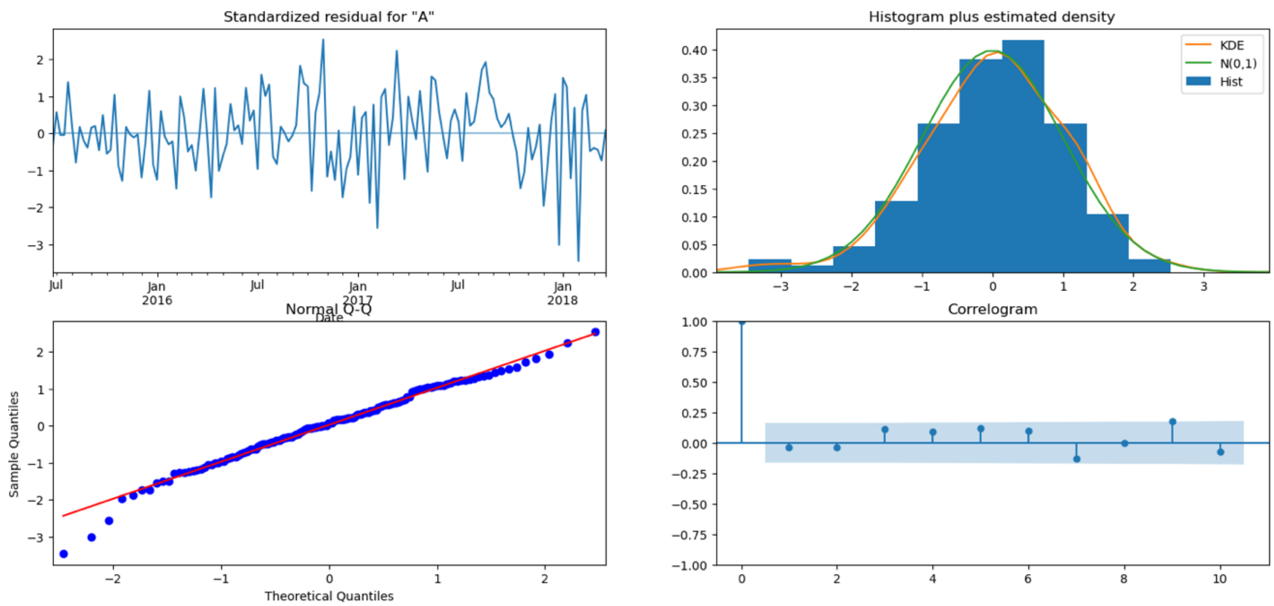
```
import statsmodels.api as sm
```

In [102…

```
mod = sm.tsa.statespace.SARIMAX(df_ar['AveragePrice'],
                                order=(1, 0, 0),
                                seasonal_order=(1, 1, 1, 12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results = mod.fit()
```

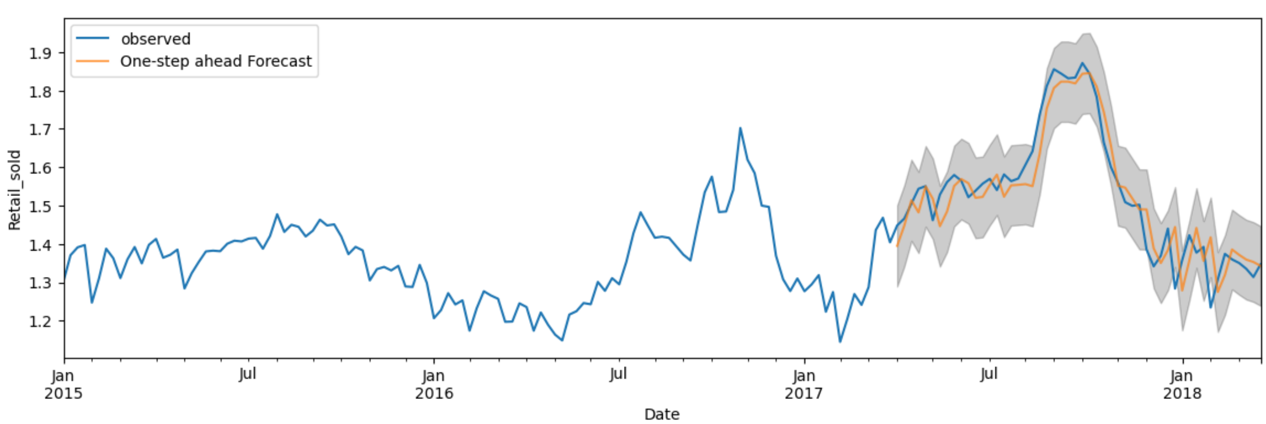
In [103…

```
results.plot_diagnostics(figsize=(18, 8))
plt.ioff()
```



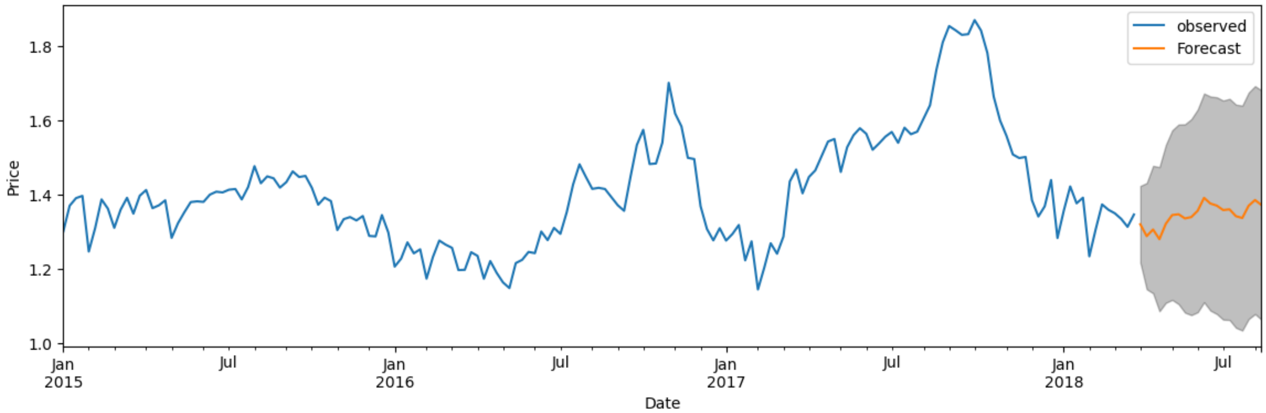
In [104…

```
pred = results.get_prediction(start=pd.to_datetime('2017-04-02'), dynamic=Fa
pred_ci = pred.conf_int()
ax = df_ar['AveragePrice'].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=.7, f
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_xlabel('Date')
ax.set_ylabel('Retail_sold')
plt.legend()
plt.show()
```



In [105…

```python
pred_uc = results.get_forecast(steps=20)
pred_ci = pred_uc.conf_int()
ax = df_ar['AveragePrice'].plot(label='observed', figsize=(14, 4))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Price')
plt.legend()
plt.show()
```



Forecasting using Facebook Prophet

Simple tutorial to install -

Install Prophet using prompt using pip:

pip install prophet

Also, we need to install plotly for plotting the data for prophet:

pip install plotly

Brief Overview gathered from various websites:

Prophet is an open-source library designed for making forecasts for univariate (one variable) time series datasets. It is easy to use and designed to automatically find a good set of hyperparameters for the model in an effort to make skillful forecasts for data with trends and seasonal structure by default. It was initially developed for the purpose of creating high quality business forecasts. It helps businesses understand and possibly predict the market. This library tries to address the following difficulties common to many business time series:

- Seasonal effects caused by human behavior: weekly, monthly and yearly cycles, dips and peaks on public holidays.
- Changes in trend due to new products and market events.
- Outliers.

It is based on a decomposable additive model where non-linear trends are fit with seasonality, it also takes into account the effects of holidays. Before we head right into coding, let's learn certain terms that are required to understand this.

Trend: The trend shows the tendency of the data to increase or decrease over a long period of time and it filters out the seasonal variations.

Seasonality: Seasonality is the variations that occur over a short period of time and is not prominent enough to be called a "trend".

Understanding the Prophet Model: The general idea of the model is similar to a generalized additive model. The "Prophet Equation" fits, as mentioned above, trend, seasonality and holidays. This is given by,

```
y(t) = g(t)+s(t)+h(t)+e(t)
```

where,

- g(t) refers to trend (changes over a long period of time)
- s(t) refers to seasonality (periodic or short term changes)
- h(t) refers to effects of holidays to the forecast
- e(t) refers to the unconditional changes that is specific to a business or a person or a circumstance. It is also called the error term.
- y(t) is the forecast.

Input to Prophet is a dataframe which must have a specific format. The first column must have the name 'ds' while the second column must have the name 'y'.

ds is ***datestamp column*** and should be as per pandas datetime format, YYYY-MM-DD or YYYY-MM-DD HH:MM:SS for a timestamp.

y is the ***numeric column we want to predict or forecast***.

This means we change the column names in the dataset. It also requires that the first column be converted to date-time objects, if they are not already.

In [106…

```
import numpy as np
import pandas as pd

from prophet import Prophet
from prophet.plot import plot_plotly
from prophet.plot import add_changepoints_to_plot

import matplotlib.pyplot as plt
```

In [107…

```
df_pr = pd.read_csv('avocado.csv', index_col=0)
df_pr['Date'] = pd.to_datetime(df_pr['Date'])
df_pr.set_index('Date', inplace=True)
```

In [108…

```
df2_week = df_pr[["AveragePrice"]].copy()
df2_week = df_pr.groupby('Date')[['AveragePrice']].mean()
df2_week.reset_index(inplace=True)
df2_week.columns = ['ds','y']
df2_week.head()
```

Out[108]:

	ds	y
0	2015-01-04	1.301296
1	2015-01-11	1.370648
2	2015-01-18	1.391111
3	2015-01-25	1.397130
4	2015-02-01	1.247037

In [109…

```
df2_week.shape
```

Out[109]:

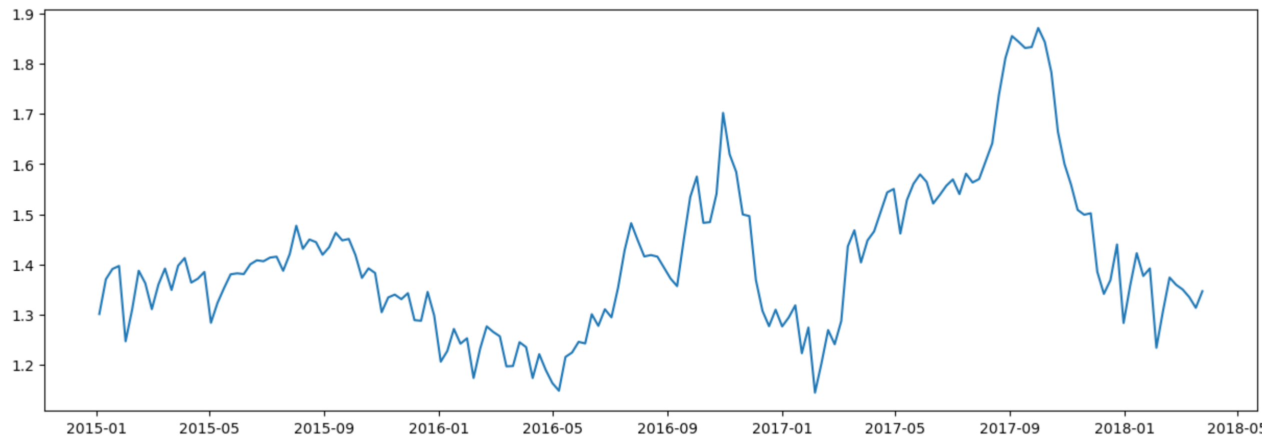
```
(169, 2)
```

In [110…

```
fig = plt.figure(figsize = (15,5))
plt.plot(df2_week.ds, df2_week.y)
```

Out[110]:

```
[<matplotlib.lines.Line2D at 0x14b736a30>]
```



In [111…

```
# Resample from weekly to monthly
df2_month = df_pr[["AveragePrice"]].resample('MS').mean()
df2_month.head()
```

Out[111]:

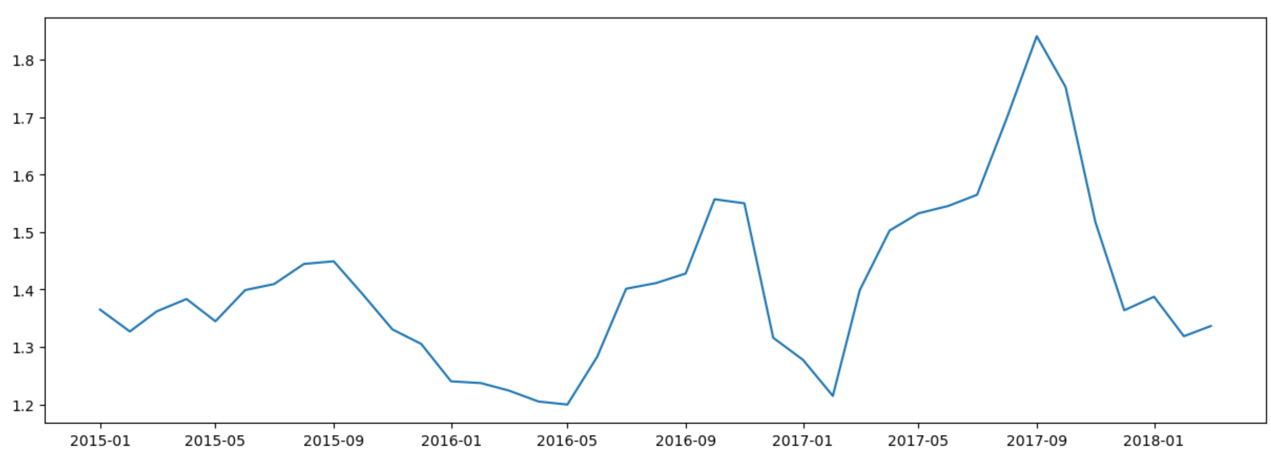
Date	AveragePrice
2015-01-01	1.365046
2015-02-01	1.326944
2015-03-01	1.361981
2015-04-01	1.383449
2015-05-01	1.344685

In [112…

```
fig = plt.figure(figsize = (15,5))
plt.plot(df2_month.index, df2_month.AveragePrice)
```

Out[112]:

```
[<matplotlib.lines.Line2D at 0x14bf5cdc0>]
```



In [113…

```
df2_month.reset_index(inplace=True)
df2_month.columns = ['ds','y']
```

Fit Prophet Model (adapted/summarized from various websites):

Prophet's API is very similar to the one you can find in sklearn. To use Prophet for forecasting, first, a Prophet( ) object is defined and configured, then it is fit on the dataset by calling the fit( ) function and passing the data, and, finally, make a forecast.

The Prophet( ) object takes arguments to configure the type of model you want, such as the type of growth, the type of seasonality, and more. By default, the model will work hard to figure out almost everything automatically.

The fit( ) function takes a DataFrame of time series data.

In [114…
```
df2_week.shape
```

Out[114]:
```
(169, 2)
```

In [115…
```
# Split data 70-30
prediction_size = 50
train_dataset = df2_week[:-prediction_size]
test_dataset = df2_week[-prediction_size:]
```

Now we need to create a new Prophet object where we can pass the parameters of the model into the constructor. Intially, we will use the defaults. Then we train our model by invoking its fit method on our training dataset:

In [116…
```
prophet_basic = Prophet()
prophet_basic.fit(train_dataset)
```

```
00:26:04 - cmdstanpy - INFO - Chain [1] start processing
00:26:05 - cmdstanpy - INFO - Chain [1] done processing
```

Out[116]:
```
<prophet.forecaster.Prophet at 0x14a9aadf0>
```

Forecasting

A forecast is made by calling the predict( ) function and passing a DataFrame that contains one column named 'ds' and rows with date-times for all the intervals to be predicted.

Using the helper method Prophet.make_future_dataframe, we create a dataframe which will contain all dates from the history and also extend into the future for those 50 weeks that we left out before.

In [117…

```
future= prophet_basic.make_future_dataframe(periods=prediction_size, freq='W

# https://rdrr.io/cran/prophet/man/make_future_dataframe.html

# How to set freq:
# https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#da
```

In [118…

```
forecast = prophet_basic.predict(future)
```

In [119…

```
forecast.head(2)
```

Out[119]:

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	addit
0	2015-01-04	1.470054	1.306641	1.440371	1.470054	1.470054	-0.093287	
1	2015-01-11	1.466338	1.312652	1.448557	1.466338	1.466338	-0.086396	

The result of the predict( ) function is a DataFrame that contains many columns. Perhaps the most important columns are the forecast date time ('ds'), the forecasted value ('yhat'), and the lower and upper bounds on the predicted value ('yhat_lower' and 'yhat_upper') that provide uncertainty of the forecast.

In [120…

```
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper', 'trend', 'trend_lower',
```

Out[120]:

	ds	yhat	yhat_lower	yhat_upper	trend	trend_lower	trend_upper
164	2018-02-25	1.462074	1.380476	1.536135	1.550943	1.513470	1.588101
165	2018-03-04	1.481569	1.402030	1.556952	1.553557	1.514713	1.592097
166	2018-03-11	1.491598	1.405167	1.566264	1.556171	1.516204	1.595968
167	2018-03-18	1.505273	1.421848	1.586587	1.558785	1.517530	1.599640
168	2018-03-25	1.526068	1.445690	1.604751	1.561399	1.519064	1.603292

Forecast quality evaluation

In [121…

```
# Create comparison dataframe
cmp_df = forecast.set_index('ds')[['yhat', 'yhat_lower', 'yhat_upper']].join
cmp_df.head()
```

Out[121]:

ds	yhat	yhat_lower	yhat_upper	y
2015-01-04	1.376768	1.306641	1.440371	1.301296
2015-01-11	1.379942	1.312652	1.448557	1.370648
2015-01-18	1.375739	1.302870	1.442015	1.391111
2015-01-25	1.351868	1.284780	1.421103	1.397130
2015-02-01	1.320043	1.252586	1.386276	1.247037

In [122...
```
cmp_df['e'] = cmp_df['y'] - cmp_df['yhat']
cmp_df['p'] = 100 * cmp_df['e'] / cmp_df['y']
```

In [123...
```
predicted_part = cmp_df[-prediction_size:]
```

MAPE is widely used as a measure of prediction accuracy because it expresses error as a percentage and thus can be used in model evaluations on different datasets. This standardizes the evaluation process.

In addition, when evaluating a forecasting algorithm, it may prove useful to calculate MAE (Mean Absolute Error) in order to have a picture of errors in absolute numbers.

In [124...
```
# Mean Absolute Percentage Error
mape = np.mean(np.abs(predicted_part['p']))
print('MAPE:', mape)
```

```
MAPE: 8.27370650546404
```

In [125...
```
# Mean Absolute Error
mae = np.mean(np.abs(predicted_part['e']))
print('MAE:', mae)
```

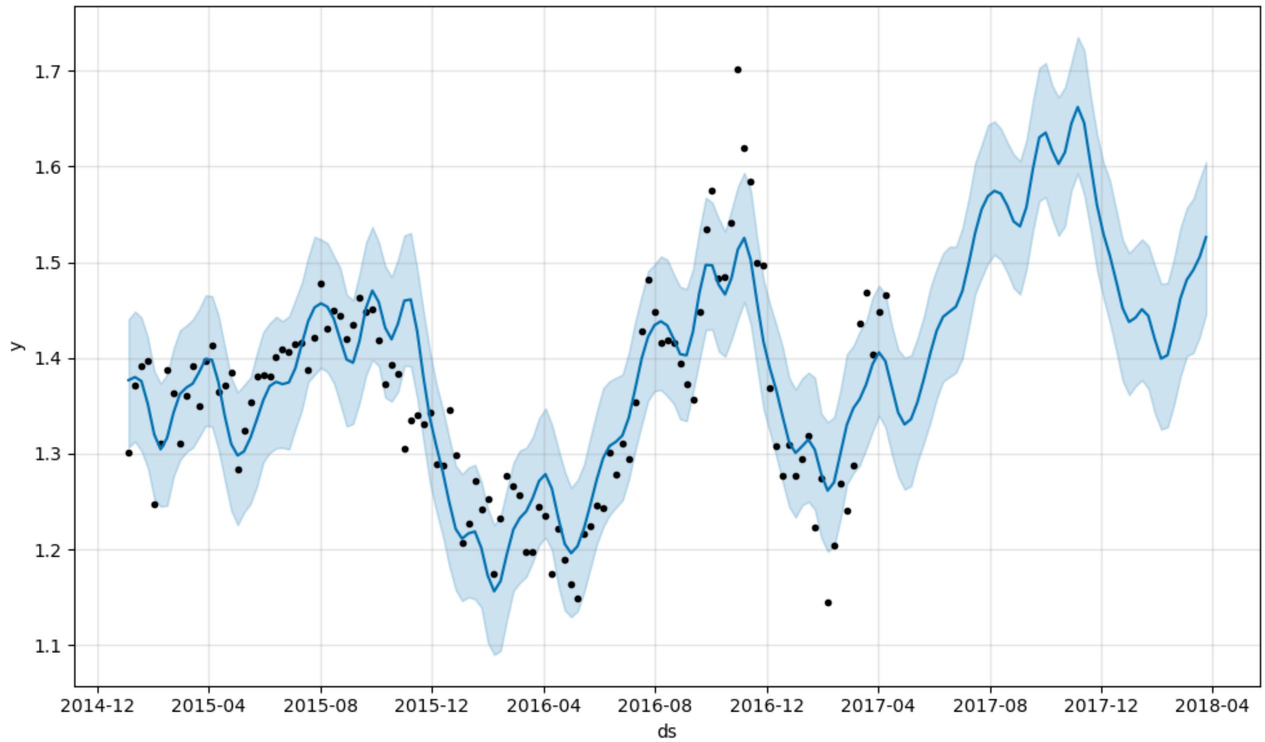
```
MAE: 0.12847575335258662
```

Visualization

The Prophet library has its own built-in tools for visualization that enable us to quickly evaluate the result.

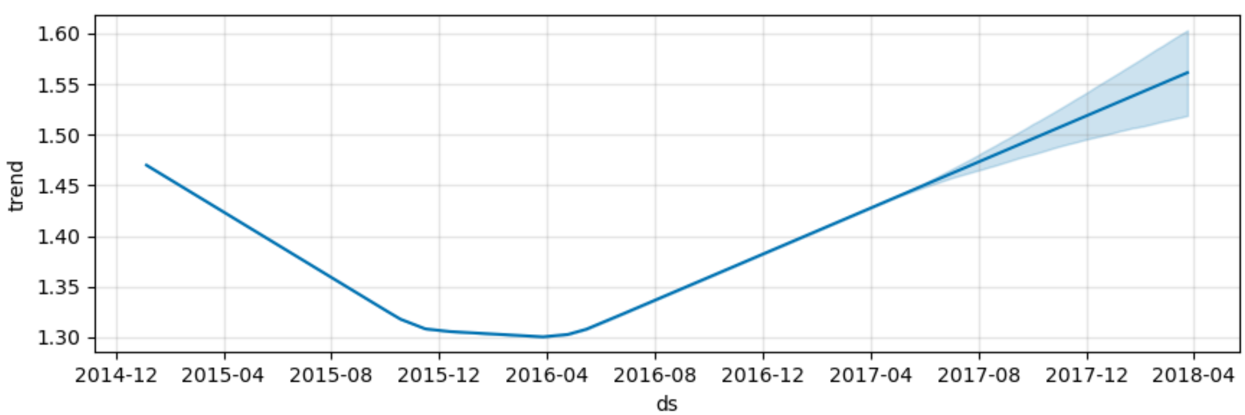
First, there is a method called Prophet.plot that will create a plot of the dataset and overlay the prediction with the upper and lower bounds for the forecast dates:

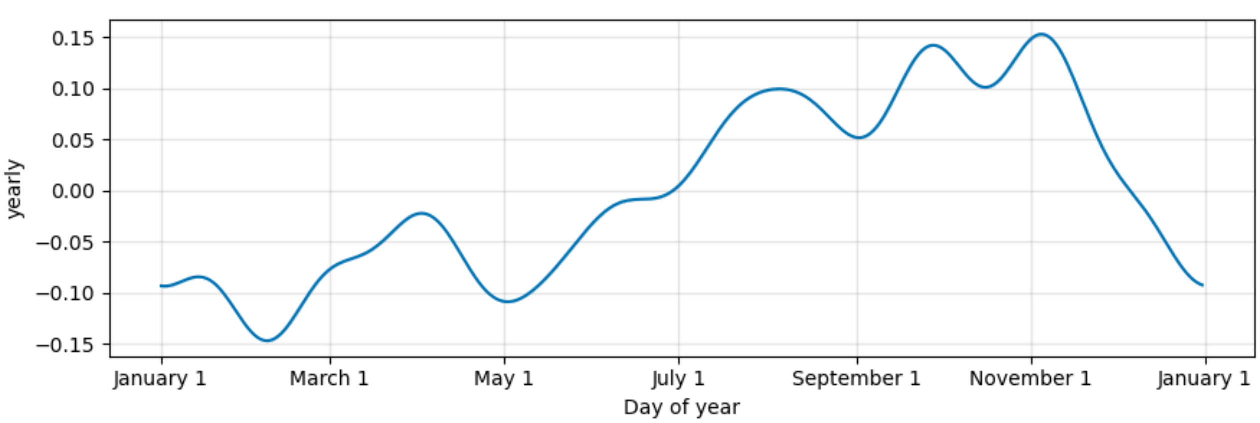
In [126...
```
fig1 = prophet_basic.plot(forecast)
```



The second function Prophet.plot_components might be much more useful in this case. It allows us to observe different components of the model separately: trend, yearly and weekly seasonality. In addition, if you supply information about holidays and events to the model, they will also be shown in the plot.

In [127…
```
fig1 = prophet_basic.plot_components(forecast)
```





The above plots shows the trends and seasonality (in a year) of the time series data. We can see there is first a decreasing and then an increasing trend, meaning the price of avocados initially dipped and then has increased over time. If we look at the seasonality graph, we can see that September to November have the highest prices at a given year. Need to check avocado farming/yield data to see if this is a supply/demand issue.

Adding ChangePoints to Prophet

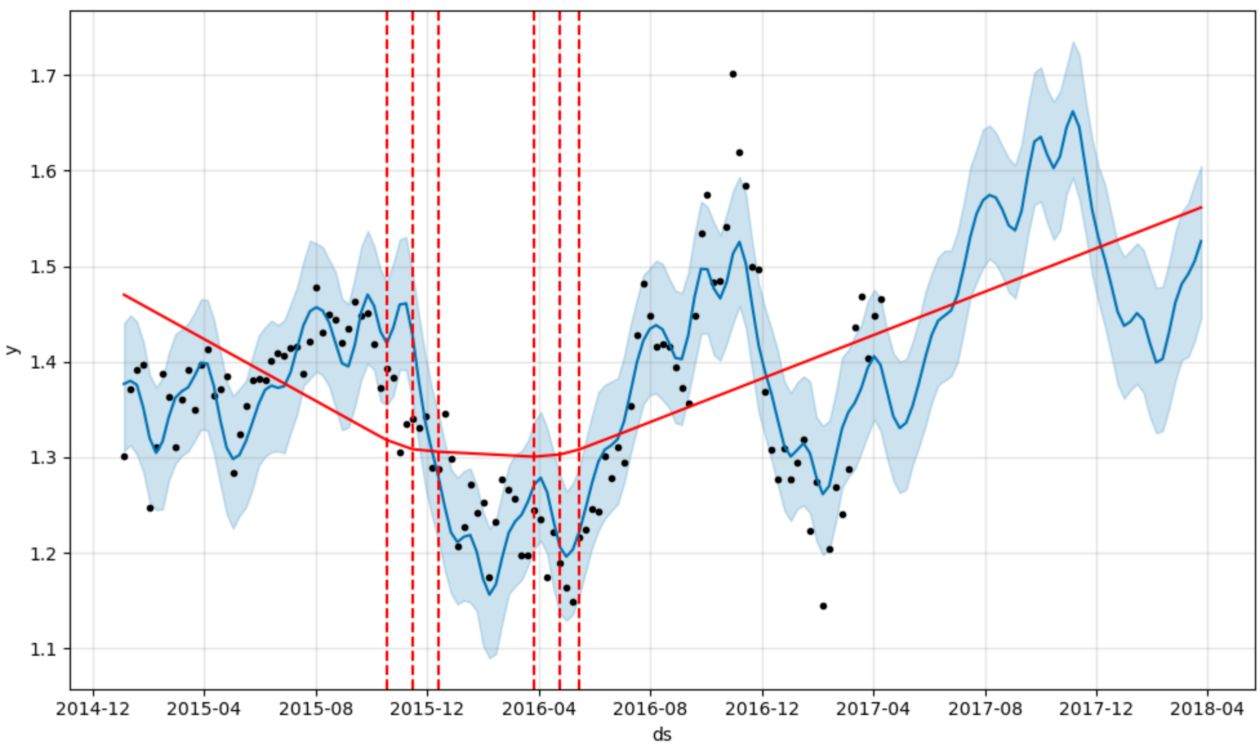
Changepoints are the datetime points where the time series have abrupt changes in the trajectory.

By default, Prophet adds 25 changepoints into the initial 80% of the dataset. The number of changepoints can be set by using the n_changepoints parameter when initializing prophet (e.g., model=Prophet(n_changepoints=30)).

Let’s plot the vertical lines where the potential changepoints occurred so we can quickly pinpoint them:

In [128…

```
fig = prophet_basic.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), prophet_basic, forecast)
```



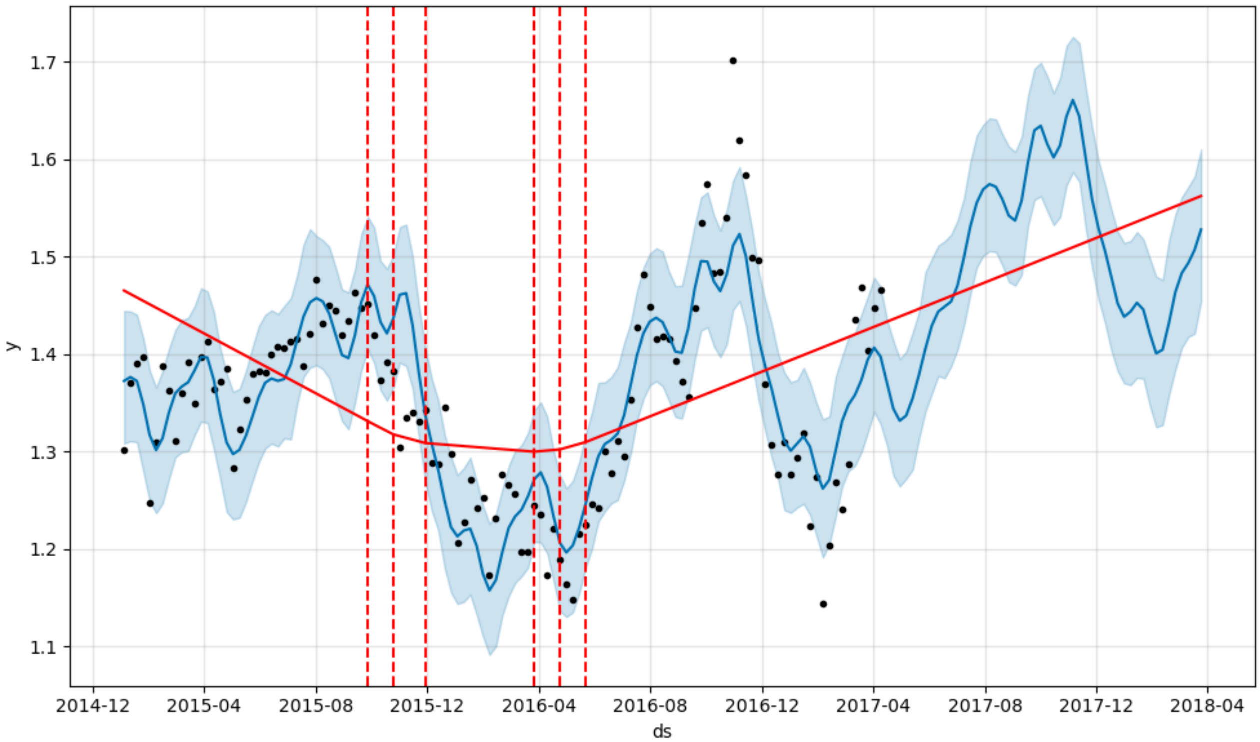
In [129…

```
# View the actual dates where the chagepoints occurred
prophet_basic.changepoints
```

Out[129]:
```
4     2015-02-01
8     2015-03-01
11    2015-03-22
15    2015-04-19
19    2015-05-17
23    2015-06-14
26    2015-07-05
30    2015-08-02
34    2015-08-30
38    2015-09-27
41    2015-10-18
45    2015-11-15
49    2015-12-13
53    2016-01-10
56    2016-01-31
60    2016-02-28
64    2016-03-27
68    2016-04-24
71    2016-05-15
75    2016-06-12
79    2016-07-10
83    2016-08-07
86    2016-08-28
90    2016-09-25
94    2016-10-23
Name: ds, dtype: datetime64[ns]
```

In [130…
```
# Change the inferred changepoint range by setting the changepoint_range
pro_change= Prophet(changepoint_range=0.9)
forecast = pro_change.fit(train_dataset).predict(future)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

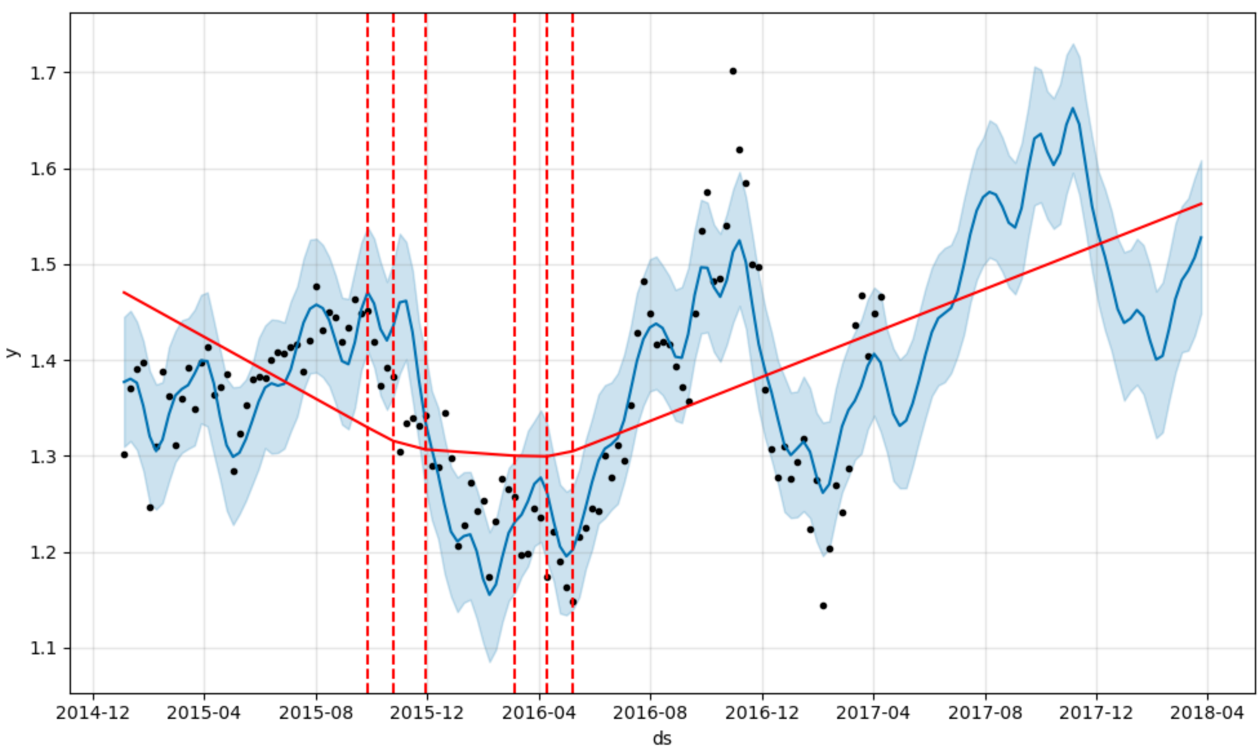
```
00:26:12 - cmdstanpy - INFO - Chain [1] start processing
00:26:12 - cmdstanpy - INFO - Chain [1] done processing
```



In [131…

```python
# The number of changepoints can be set by using the n_changepoints paramete
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True)
forecast = pro_change.fit(train_dataset).predict(future)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

```
00:26:16 - cmdstanpy - INFO - Chain [1] start processing
00:26:16 - cmdstanpy - INFO - Chain [1] done processing
```



Adjusting Trend

Prophet allows you to adjust the trend in case there is an overfit (too much flexibility) or underfit (not enough flexibility).

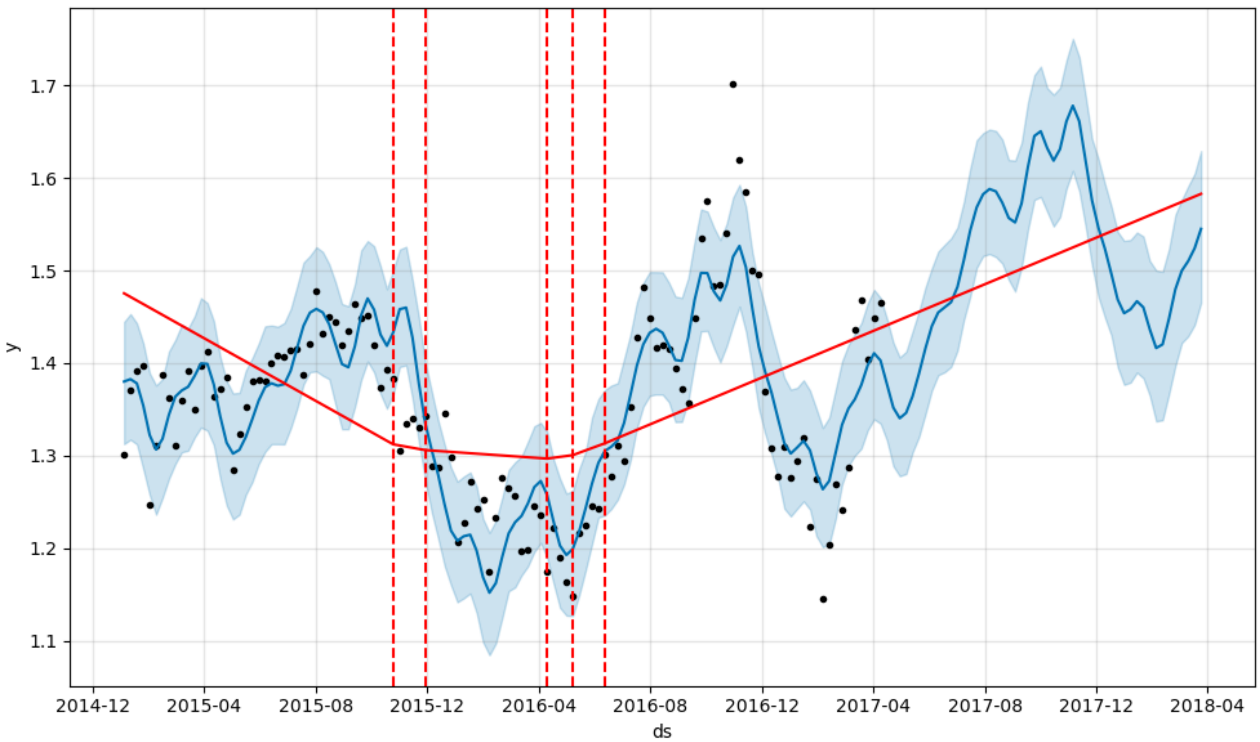
changepoint_prior_scale helps adjust the strength of the trend. The default value for changepoint_prior_scale is 0.05.

We can decrease the value to make the trend less flexible or we can increase the value of changepoint_prior_scale to make the trend more flexible.

In [132…

```
# Increasing the changepoint_prior_scale to 0.08 to make the trend flexible
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True, changepoint_
forecast = pro_change.fit(train_dataset).predict(future)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

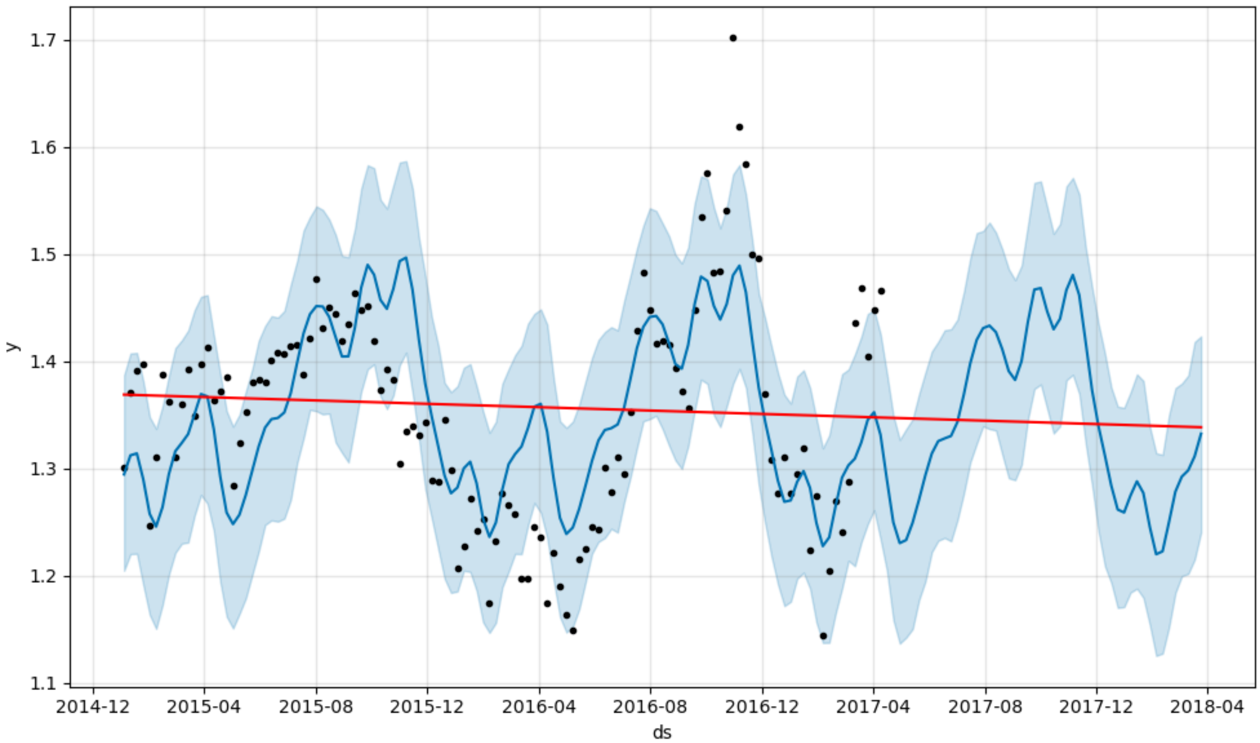
```
00:26:19 - cmdstanpy - INFO - Chain [1] start processing
00:26:19 - cmdstanpy - INFO - Chain [1] done processing
```



In [133…

```
# Decreasing the changepoint_prior_scale to 0.001 to make the trend less fle
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True, changepoint_
forecast = pro_change.fit(train_dataset).predict(future)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

```
00:26:21 - cmdstanpy - INFO - Chain [1] start processing
00:26:21 - cmdstanpy - INFO - Chain [1] done processing
00:26:21 - cmdstanpy - ERROR - Chain [1] error: error during processing Stal
e NFS file handle
Optimization terminated abnormally. Falling back to Newton.
00:26:22 - cmdstanpy - INFO - Chain [1] start processing
00:26:22 - cmdstanpy - INFO - Chain [1] done processing
```



Adding Holidays

Holidays and events can cause changes to a time series. In our example the National Avocado day on July 31 and Guacamole day on September 16 may have impacted the prices of the Avocado.

Create a custom holiday list for Prophet by creating a dataframe with two columns 'ds' and 'holiday'. A row for each occurrence of the holiday:

In [134…

```
avocado_season = pd.DataFrame({
  'holiday': 'avocado season',
  'ds': pd.to_datetime(['2014-07-31', '2014-09-16',
                        '2015-07-31', '2015-09-16',
                        '2016-07-31', '2016-09-16',
                        '2017-07-31', '2017-09-16',
                       '2018-07-31', '2018-09-16',
                        '2019-07-31', '2019-09-16']),
  'lower_window': -1, # to include the day before the holiday
  'upper_window': 0,
})
```

In [135…

```
avocado_season
```

Out[135]:

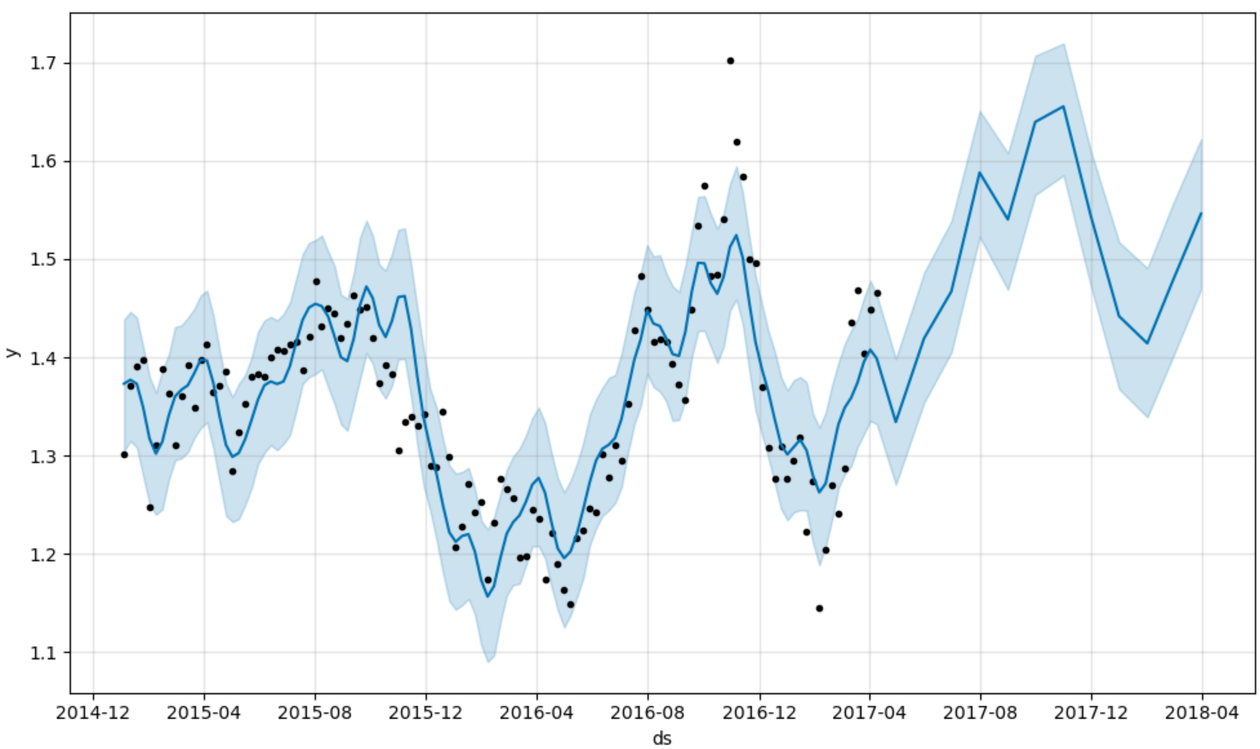
	holiday	ds	lower_window	upper_window
0	avocado season	2014-07-31	-1	0
1	avocado season	2014-09-16	-1	0
2	avocado season	2015-07-31	-1	0
3	avocado season	2015-09-16	-1	0
4	avocado season	2016-07-31	-1	0
5	avocado season	2016-09-16	-1	0
6	avocado season	2017-07-31	-1	0
7	avocado season	2017-09-16	-1	0
8	avocado season	2018-07-31	-1	0
9	avocado season	2018-09-16	-1	0
10	avocado season	2019-07-31	-1	0
11	avocado season	2019-09-16	-1	0

In [136…

```
pro_holiday= Prophet(holidays=avocado_season)
pro_holiday.fit(train_dataset)
future_data = pro_holiday.make_future_dataframe(periods=12, freq = 'm')

# Forecast the data for future data
forecast_data = pro_holiday.predict(future_data)
fig1 = pro_holiday.plot(forecast_data)
```

```
00:26:24 - cmdstanpy - INFO - Chain [1] start processing
00:26:24 - cmdstanpy - INFO - Chain [1] done processing
```



The forecast predicts a generally increasing trend in price.

Adding Multiple Regressors

Adding regressors to get more granular trends. (Important: Additional regressor column value needs to be present in both the fitting as well as prediction dataframes)

In [137…

```
df2_week['type'] = dataset['type']
df2_week['Total Volume'] = dataset['Total Volume']
df2_week['4046'] = dataset['4046']
df2_week['4225'] = dataset['4225']
df2_week['4770'] = dataset['4770']
df2_week['Small Bags'] = dataset['Small Bags']
```

In [138…

```
df2_week.shape
```

Out[138]:

```
(169, 8)
```

In [139…

```
train_X= df2_week[:-prediction_size]
test_X= df2_week[-prediction_size:]
```

In [140…

```
# Additional Regressor
pro_regressor= Prophet()
pro_regressor.add_regressor('Total Volume')
pro_regressor.add_regressor('4046')
pro_regressor.add_regressor('4225')
pro_regressor.add_regressor('4770')
pro_regressor.add_regressor('Small Bags')
```

Out[140]:

```
<prophet.forecaster.Prophet at 0x148ac99a0>
```

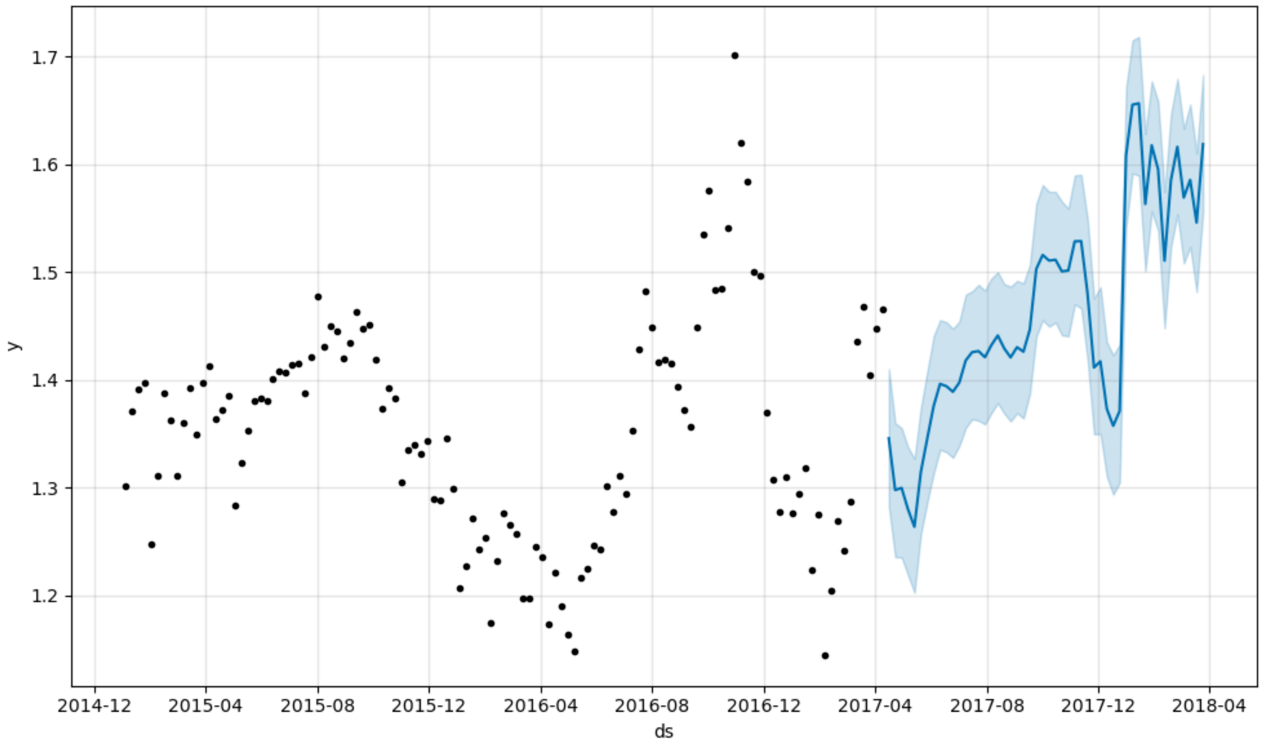
In [141…

```
# Fitting the data
pro_regressor.fit(train_X)
future_data = pro_regressor.make_future_dataframe(periods=365)
```

```
00:26:28 - cmdstanpy - INFO - Chain [1] start processing
00:26:28 - cmdstanpy - INFO - Chain [1] done processing
```

In [142…

```
# Forecast the data for test  data
forecast_data = pro_regressor.predict(test_X)
fig1 = pro_regressor.plot(forecast_data)
```



1.7
1.6
1.5
1.4
1.3
1.2
y
2014-12
2015-04
2015-08
2015-12
2016-04
2016-08
2016-12
2017-04
2017-08
2017-12
2018-04
ds